ECMAScript (ES6+)

ES6 (ECMAScript 2015) introduced new syntax as "syntax sugar" - cleaner ways to write JavaScript

transpilers: convert new JS to legacy code for older browsers (e.g., Babel) most modern browsers have built-in support, but we still transpile for full compatibility

---

Variables: var vs let vs const

Feature	`var`	`let`	`const`
Scope	function	block	block
Hoisting	hoisted (undefined)	hoisted (throws error)	hoisted (throws error)
Re-declaration	✅ allowed	❌ error	❌ error
Re-assignment	✅ yes	✅ yes	❌ no
Initial value	optional	optional	⚠ must initialize

temporal dead zone (TDZ):

`let` and `const` are hoisted but NOT initialized - accessing them before declaration throws an error

```
let x = 10;
function test() {
  console.log(x); // ❌ ReferenceError! x is in TDZ
  let x = 20; // x is hoisted but not initialized yet
}
test();
console.log(x); // 10 (outer x)
```

⚠ this is called the "temporal dead zone" - the variable exists but can't be accessed until initialization

---

Template Literals

using backticks `` ` `` instead of quotes - allows string interpolation and multi-line strings

```
// old way (concatenation)
var name = "Ahmed";
var msg = "Hello, " + name + "! You are " + age + " years old.";

// new way (template literals)
var msg = `Hello, ${name}! You are ${age} years old.`;

// multi-line strings
var html = `
  <div>
    <h1>${title}</h1>
    <p>${content}</p>
  </div>
`;
```

Arrow Functions

shorter syntax for anonymous functions

```
// old way
var add = function (a, b) {
  return a + b;
};

// arrow function
var add = (a, b) => {
  return a + b;
};

// one line - can remove {} and return
var add = (a, b) => a + b;

// one argument - can remove ()
var double = (x) => x * 2;

// no arguments - need empty ()
var greet = () => "hello";
```

⚠ **arrow functions DON'T have their own `this`:**

```
var obj = {
  name: "Ahmed",

  // regular function - has its own 'this'
  sayHi: function () {
    console.log(this.name); // "Ahmed"
  },

  // arrow function - inherits 'this' from parent scope
  sayHiArrow: () => {
    console.log(this.name); // undefined! 'this' is window
  },
};
```

⚠ use regular functions for object methods, arrow functions for callbacks

---

Destructuring

extract values from arrays/objects into variables in one line

array destructuring:

```
var arr = [1, 3];

// old way
var x = arr[0];
var y = arr[1];

// new way (destructuring)
let [x, y] = arr; // x = 1, y = 3
let [x, y, z] = arr; // z = undefined (doesn't exist)

// can initialize directly
let [a, b, c] = [1, 2, "asaad"]; // c = "asaad"

// skip values with empty slots
let [first, , third] = [1, 2, 3]; // first = 1, third = 3

// from function return
function test() {
  return [1, 2];
}
let [x, y] = test(); // x = 1, y = 2
```

object destructuring:

```
// old way
let x = window.name;

// new way - extract and rename
let { name: x, location: y } = window; // x = window.name, y = window.location

// if variable name is same as property name
let { name, location } = window; // name = window.name, location = window.location

// default values (if property doesn't exist)
let { name, foo = "default" } = window; // foo = "default" if not in window
```

⚠ if attribute does not exist in the object, it will be `undefined` (unless you set a default)

Spread and Rest Operator (...)

the `...` operator works differently based on where it's used

rest parameter (in function parameters): collects remaining arguments into an array

```
function test(...x) {
  console.log(x);
}
test(1); // [1]
test(1, 2, 3, 4); // [1, 2, 3, 4]
test(); // []

// with other parameters - rest MUST be last
function test(a, ...rest) {
  console.log(a); // first argument
  console.log(rest); // remaining as array
}
test(1, 2, 3, 4); // a = 1, rest = [2, 3, 4]
test(); // a = undefined, rest = []
```

⚠ rest parameter must be the LAST parameter

spread operator (expanding arrays/objects): spreads elements into individual values

```
var arr1 = [1, 2, 3];
var arr2 = [4, 5, 6];

// combining arrays
var arr3 = [...arr1, ...arr2]; // [1, 2, 3, 4, 5, 6]
var arr4 = [...arr1, arr2]; // [1, 2, 3, [4, 5, 6]] - without spread!

// passing array elements as function arguments
Math.max(arr1); // NaN (can't take array)
Math.max(...arr1); // 3 (spreads as Math.max(1, 2, 3))

// copying arrays (shallow copy, but copies primitives correctly)
var original = [1, 2, 3];
var copy = original; //  same reference, not a copy!
var realCopy = [...original]; //  new array with copied values

// spread with objects (ES2018)
var obj1 = { a: 1, b: 2 };
var obj2 = { ...obj1, c: 3 }; // { a: 1, b: 2, c: 3 }
```

⚠ spread creates a **shallow copy** - nested objects/arrays are still references

---

User Defined Objects

objects are key-value pairs that can hold properties and methods all objects inherit from `Object` (the parent of all objects in JavaScript)

creating objects:

```
// object literal (most common)
let obj = {};

// using Object constructor
let obj = new Object();

// with initial properties
let obj = { name: "asaad", age: 22 };
```

adding/modifying properties:

```
let obj = { name: "asaad" };

// add new property
obj.age = 22;

// overwrite existing property
obj.age = 21;

// add nested objects
obj.address = { country: "egypt", city: "cairo" };
```

accessing properties:

```
// dot notation
console.log(obj.name);           // "asaad"
console.log(obj.address.country); // "egypt"

// bracket notation (subscribe notation)
console.log(obj["age"]);         // 22

// useful when property name is in a variable
let key = "age";
console.log(obj[key]);           // 22
obj[key] = 60;                   // updates age to 60
```

⚠ accessing a non-existent property returns `undefined`

```
console.log(obj.eyes); // undefined (doesn't exist)
```

storing functions in objects:

```
// object can hold functions (methods)
obj.print = (x) => console.log(x);

// calling the method
obj.print("hello"); // "hello"

// or define inline
let person = {
  name: "asaad",
  greet: function() {
    console.log("Hello, " + this.name);
  }
};
person.greet(); // "Hello, asaad"
```

deleting properties:

```
delete obj.name;
console.log(obj.name); // undefined (deleted)
```

looping over object properties:

```
let obj = { name: "asaad", age: 22, city: "cairo" };

// for...in loop - iterates over keys
for (let key in obj) {
  console.log(key);       // "name", "age", "city" (strings)
  console.log(obj[key]);  // "asaad", 22, "cairo" (values)
}
```

⚠ use `obj[key]` not `obj.key` in loops - `obj.key` looks for a property literally named "key"

Changing `this` with call, apply, and bind

these methods let you change what `this` refers to when calling a function

Method	Description	Arguments
`call()`	calls function immediately with new `this`	`(thisArg, arg1, arg2...)`
`apply()`	same as call, but arguments as array	`(thisArg, [arg1, arg2])`
`bind()`	returns new function with bound `this` (no call)	`(thisArg, arg1, arg2...)`

borrowing array methods for strings:

```
var str = "asaad";

// strings don't have forEach, but arrays do
// borrow forEach from array and use string as 'this'
[].forEach.call(str, function(char) {
  console.log(char); // "a", "s", "a", "a", "d"
});

// same with apply (arguments as array)
[].forEach.apply(str, [function(char) {
  console.log(char);
}]);

// bind returns a new function for later use
let forEachOnStr = [].forEach.bind(str);
forEachOnStr(function(char) {
  console.log(char);
});
```

changing `this` in object methods:

```
let person = {
  name: "asaad",
  age: 21,
  print: function() {
    console.log(person.name, person.age); // hardcoded - won't work with call
  }
};

let employee = {
  name: "amr",
  age: 22
};

person.print.call(employee); // still prints "asaad", 21 (hardcoded!)
```

making methods generic with `this`:

```
let person = {
  name: "asaad",
  age: 21,
  print: function() {
    console.log(this.name, this.age); // uses 'this' - now it's generic!
  }
};

let employee = {
  name: "amr",
  age: 22
};

person.print();              // "asaad", 21 (this = person)
person.print.call(employee); // "amr", 22 (this = employee)
```

the `this` problem with callbacks:

```
let person = {
  name: "asaad",
  age: 21,
  init: function() {
    // inside this function, 'this' = person

    document.querySelector("div").addEventListener("click", function() {
      // ❌ inside callback, 'this' = the div element, NOT person!
      console.log(this.name, this.age); // undefined, undefined
    });
  }
};
```

solutions for `this` in callbacks:

```
// solution 1: arrow function (inherits 'this' from parent scope)
let person = {
  name: "asaad",
  age: 21,
  init: function() {
    document.querySelector("div").addEventListener("click", () => {
      console.log(this.name, this.age); // ✅ "asaad", 21
    });
  }
};

// solution 2: bind() to preserve 'this'
let person = {
  name: "asaad",
  age: 21,
  init: function() {
    setTimeout(function() {
      console.log(this.name, this.age); // ✅ "asaad", 21
    }.bind(this), 1000);
  }
};

// solution 3: store 'this' in a variable (older pattern)
let person = {
  name: "asaad",
  age: 21,
  init: function() {
    var self = this; // save reference
    setTimeout(function() {
      console.log(self.name, self.age); // ✅ "asaad", 21
    }, 1000);
  }
};
```

⚠ arrow functions don't have their own `this` - they inherit from parent scope, making them perfect for callbacks

---

Constructor Functions and the `new` Keyword

"functions are first-class objects" - in JavaScript, functions are objects that can have properties and methods

the `new` keyword is the magic:

```
// without 'new' - just a regular function call
function Employee() {
  console.log(this); // 'this' refers to window (the caller)
}
Employee(); // logs: Window object

// with 'new' - creates a new object!
function Employee(n, a) {
  console.log(this); // 'this' refers to the newly created object!
  this.name = n;
  this.age = a;
  // implicitly returns {} (the new object)
}

let emp = new Employee("asaad", 21);
// emp is now an object of type Employee with a constructor!
console.log(emp.name); // "asaad"
console.log(emp.age);  // 21
```

what `new` does behind the scenes:

1. creates a new empty object `{}`
2. sets `this` to point to that new object
3. links the object to the constructor's prototype
4. implicitly returns the object

Prototypes

functions are objects - they have properties we can access with dot notation

every function has a special property called `prototype` which is itself an `Object` with a `constructor` property

```
// JavaScript automatically creates:
// Employee.prototype = { constructor: Employee }

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

// the problem with methods inside constructor:
function Employee(name, age) {
  this.name = name;
  this.age = age;
  this.print = function() {
    console.log(this.name, this.age);
  };
  // ⚠ BAD PERFORMANCE: each object gets its own copy of print() in memory!
}
```

solution: use prototype for shared methods

just like arrays have `.length` on their prototype (not on each instance), we can add methods to prototype

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

// add method to prototype - shared by ALL instances!
Employee.prototype.print = function() {
  console.log(this.name, this.age);
};

let emp1 = new Employee("asaad", 21);
let emp2 = new Employee("ahmed", 25);

emp1.print(); // "asaad", 21
emp2.print(); // "ahmed", 25

// both emp1 and emp2 share the SAME print function
console.log(emp1.print === emp2.print); // true  (same reference)
```

how prototype lookup works:

```
// when you call emp1.print():
// 1. JavaScript looks for 'print' on emp1 object - not found
// 2. JavaScript looks in emp1's prototype (Employee.prototype) - found!
// 3. executes the function with 'this' set to emp1

// you can also add to built-in prototypes (use carefully!)
Array.prototype.first = function() {
  return this[0];
};
[1, 2, 3].first(); // 1
```

⚠ methods on prototype = better performance (shared memory) ⚠ methods in constructor = each instance has its own copy (more memory)

Factory Functions

factory functions create and return new objects - useful for creating multiple similar objects

```
// factory function
function createPerson(name, age) {
  return {
    name: name,
    age: age,
    greet: function() {
      console.log("Hi, I'm " + this.name);
    }
  };
}

// create multiple objects
let person1 = createPerson("asaad", 22);
let person2 = createPerson("ahmed", 25);

person1.greet(); // "Hi, I'm asaad"
person2.greet(); // "Hi, I'm ahmed"
```

shorthand property syntax (ES6): if variable name matches property name

```
function createPerson(name, age) {
  return {
    name,  // same as name: name
    age,   // same as age: age
    greet() {  // shorthand method syntax
      console.log("Hi, I'm " + this.name);
    }
  };
}
```

factory vs constructor functions:

Feature	Factory Function	Constructor Function
Syntax	`createPerson()`	`new Person()`
Returns	explicit `return {}`	implicit return
`this`	not used (or explicitly)	refers to new object
Prototype methods	not easy	`Person.prototype.method`
`instanceof`	doesn't work	works

ES6 Classes

ECMAScript made classes as **syntactic sugar** over constructor functions - cleaner syntax, same behavior

basic class syntax:

```
class Employee {
  // constructor - called when using 'new'
  constructor(n, a, s) {
    this.name = n;
    this.age = a;
    this.salary = s;
  }
}

let emp = new Employee("asaad", 22, 21111);
console.log(emp.name); // "asaad"
```

adding validation in constructor:

```
class Employee {
  constructor(n, a, s) {
    // type checking
    if (typeof n !== "string") {
      throw new Error("Name must be a string!");
    }
    this.name = n;
    this.age = a;
    this.salary = s;
  }
}

new Employee(123, 22, 5000); //  Error: Name must be a string!
```

no method overloading in JavaScript:

JavaScript doesn't support multiple constructors with different parameters - we can already pass any number of arguments

solution: use default parameter values

```
class Employee {
  constructor(n, a = 0, s = 0) {  // default values for age and salary
    if (typeof n !== "string") {
      throw new Error("Name must be a string!");
    }
    this.name = n;
    this.age = a;
    this.salary = s;
  }
}

let emp1 = new Employee("asaad", 22, 5000); // all params
let emp2 = new Employee("ahmed", 25);       // salary defaults to 0
let emp3 = new Employee("sara");            // age and salary default to 0
```

Private Fields and Getters/Setters

private fields with `#`:

prefix with `#` to make a field private - cannot be accessed outside the class

```
class Employee {
  #name;  // private field - must declare before using
  #salary;

  constructor(n, a, s) {
    this.#name = n;   // private
    this.age = a;     // public
    this.#salary = s; // private
  }
}

let emp = new Employee("asaad", 22, 5000);
console.log(emp.age);     // 22 ✅ (public)
console.log(emp.name);    // undefined ❌ (no public 'name' property)
console.log(emp.#name);   // ❌ SyntaxError! private field
```

getters and setters:

use `get` and `set` to control access to properties - they're accessed like properties, not called like functions!

```
class Employee {
  #name;

  constructor(n, a) {
    this.#name = n;
    this.age = a;
  }

  // getter - access with emp.name (no parentheses!)
  get name() {
    return this.#name;
  }

  // setter - set with emp.name = "value" (no parentheses!)
  set name(value) {
    if (typeof value !== "string") {
      throw new Error("Name must be a string!");
    }
    this.#name = value;
  }
}

let emp = new Employee("asaad", 22);
console.log(emp.name);  // "asaad" (calls getter)
emp.name = "ahmed";     // calls setter
console.log(emp.name);  // "ahmed"
emp.name = 123;         // □ Error: Name must be a string!
```

⚠ getters/setters are added to the **prototype** as properties, not in the constructor

---

Class Methods

methods defined in class go to prototype automatically:

```
class Employee {
  #name;

  constructor(n, a) {
    this.#name = n;
    this.age = a;
  }

  // method - automatically added to Employee.prototype!
  print() {
    console.log(this.#name, this.age);
  }

  // another method
  giveRaise(amount) {
    this.salary += amount;
  }
}

let emp1 = new Employee("asaad", 22);
let emp2 = new Employee("ahmed", 25);

emp1.print(); // "asaad", 22
emp2.print(); // "ahmed", 25

// both share the same method (good for memory!)
console.log(emp1.print === emp2.print); // true □
```

⚠ JavaScript automatically adds class methods to the prototype (not to each instance) - better memory performance!

Static Methods and Properties

static methods belong to the class itself, not instances:

```
class Employee {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  // instance method - goes to prototype
  print() {
    console.log(this.name, this.age);
  }

  // static method - belongs to Employee class itself
  static createAnonymous() {
    return new Employee("Anonymous", 0);
  }
}

let emp = new Employee("asaad", 22);
emp.print();              // ✅ works (instance method)
emp.createAnonymous();    // ❌ Error! instances don't see static methods

Employee.createAnonymous(); // ✅ works (called on class)
```

how static works natively (without `class` syntax):

```
 // static methods are just properties on the constructor function itself
function Employee(name) {
  this.name = name;
}

// instance method - on prototype (instances can see it)
Employee.prototype.print = function() {
  console.log(this.name);
};

// static method - on the function object itself (instances can't see it)
Employee.createAnonymous = function() {
  return new Employee("Anonymous");
};

let emp = new Employee("asaad");
emp.print();              // ✅ works
Employee.createAnonymous(); // ✅ works
emp.createAnonymous();    // ❌ undefined (instances only see prototype)
```

⚠ instances only see **prototype** - static methods are on the class/constructor object itself

creating a "static class" (utility class):

to prevent instantiation, throw an error in the constructor

```
class MathUtils {
  constructor() {
    throw new Error("MathUtils cannot be instantiated!");
  }

  static add(a, b) {
    return a + b;
  }

  static multiply(a, b) {
    return a * b;
  }
}

MathUtils.add(5, 3);      // 8 
new MathUtils();          //  Error: MathUtils cannot be instantiated!
```

Inheritance with `extends`

use `extends` to inherit from a parent class, and `super()` to call the parent constructor

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

  greet() {
    console.log(`Hi, I'm ${this.name}`);
  }
}

class Employee extends Person {
  constructor(name, age, salary) {
    super(name, age);  // MUST call parent constructor first!
    this.salary = salary;
  }

  // override parent method
  greet() {
    console.log(`Hi, I'm ${this.name} and I work here`);
  }

  // call parent method with super
  greetFormal() {
    super.greet();  // calls Person's greet()
    console.log(`My salary is ${this.salary}`);
  }
}

let emp = new Employee("asaad", 22, 5000);
emp.greet();        // "Hi, I'm asaad and I work here" (overridden)
emp.greetFormal();  // "Hi, I'm asaad" then "My salary is 5000"

console.log(emp instanceof Employee); // true
console.log(emp instanceof Person);   // true (inheritance chain)
```

⚠ in child constructor, you **MUST call `super()` before using `this`**

Prototype Chaining and Method Override

every function's prototype points to Object:

```
function Employee(name) {
  this.name = name;
}

// prototype chain:
// emp → Employee.prototype → Object.prototype → null

console.log(Employee.prototype.constructor); // Employee
console.log(Object.getPrototypeOf(Employee.prototype)); // Object.prototype
```

how method override works - "the closer function wins":

JavaScript looks for methods starting from the object, then up the prototype chain - the **first match** is used

```
class Person {
  greet() {
    console.log("Hello from Person");
  }
}

class Employee extends Person {
  // override - this is "closer" to the instance
  greet() {
    console.log("Hello from Employee");
  }
}

let emp = new Employee();
emp.greet(); // "Hello from Employee" (closer wins!)

// lookup order:
// 1. emp object itself - not found
// 2. Employee.prototype - FOUND! uses this one
// 3. Person.prototype - never reached (already found above)
```

⚠ to override, just define the method again in the child class - the "closer" one gets called first (prototype chaining)

Abstract Classes

JavaScript doesn't have true abstract classes, but we can simulate them by checking `this.constructor`

simulating abstract class:

```
class Shape {
  constructor() {
    // prevent direct instantiation of Shape
    if (this.constructor === Shape) {
      throw new Error("Shape is abstract - cannot instantiate directly!");
    }
  }

  // "abstract" method - should be overridden
  getArea() {
    throw new Error("getArea() must be implemented by subclass!");
  }
}

class Rectangle extends Shape {
  constructor(width, height) {
    super();  // calls Shape constructor, but this.constructor === Rectangle ✅
    this.width = width;
    this.height = height;
  }

  // override the abstract method
  getArea() {
    return this.width * this.height;
  }
}

new Shape();              // ❌ Error: Shape is abstract!
let rect = new Rectangle(5, 10);  // ✅ works (this.constructor === Rectangle)
rect.getArea();           // 50 ✅
```

why it works:

- when `new Shape()` is called → `this.constructor === Shape` → throws error ❌
- when `new Rectangle()` is called → `super()` runs Shape's constructor, but `this.constructor === Rectangle` → allowed ✅

⚠ child classes call `super()` but their `this.constructor` points to themselves, not the parent