Object-Oriented Programming (OOP)

> 🎓 **CS Perspective:** JavaScript implements OOP through **prototypal inheritance** rather than classical inheritance (like Java/C++). Instead of classes as blueprints, objects inherit directly from other objects via the **prototype chain**—a linked list of objects searched during property lookup. ES6 classes are **syntactic sugar** over this prototype system. The `new` keyword implements a specific algorithm: object creation, prototype linking, and implicit return. Understanding the difference between `__proto__` (instance's prototype link) and `.prototype` (constructor's prototype property) is crucial. This prototypal model is more flexible than classical OOP, enabling patterns like **object composition** and **mixins** without class hierarchies.

📌 OOP Principles

Principle	Description
Encapsulation	Bundling data + methods, hiding internals
Abstraction	Hiding complexity, exposing simple interface
Inheritance	Child class inherits from parent class
Polymorphism	Child class can override parent methods

📌 Constructor Functions

Traditional way to create objects (before ES6 classes):

```
const Person = function(firstName, birthYear) {
  // Instance properties
  this.firstName = firstName;
  this.birthYear = birthYear;

  // ❌ Never add methods here (created for each instance)
  // this.calcAge = function() { ... }
};

const jonas = new Person('Jonas', 1991);
console.log(jonas instanceof Person); // true
```

What `new` Does:

1. Creates empty object `{}`
2. Sets `this` to the new object
3. Links object to prototype
4. Returns object automatically

📌 Prototypes

Add methods to prototype (shared by all instances):

```
Person.prototype.calcAge = function() {
  console.log(2037 - this.birthYear);
};

jonas.calcAge(); // 46

// Check prototype
console.log(jonas.__proto__ === Person.prototype); // true
console.log(Person.prototype.isPrototypeOf(jonas)); // true
```

Prototype Chain

```
jonas.__proto__;                // Person.prototype
jonas.__proto__.__proto__;      // Object.prototype
jonas.__proto__.__proto__.__proto__; // null (end of chain)
```

Own vs Inherited Properties

```
jonas.hasOwnProperty('firstName'); // true
jonas.hasOwnProperty('calcAge');   // false (inherited)
```

📌 ES6 Classes

Syntactic sugar over constructor functions:

```
class PersonCl {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
    this.birthYear = birthYear;
  }

  // Instance methods (on prototype)
  calcAge() {
    console.log(2037 - this.birthYear);
  }

  // Getter
  get age() {
    return 2037 - this.birthYear;
  }
  // Setter (validation)
  set fullName(name) {
    if (name.includes(' ')) this._fullName = name;
    else alert('Not a full name!');
  }

  get fullName() {
    return this._fullName;
  }

  // Static method (on class itself)
  static hey() {
    console.log('Hey there 👋');
  }
}

const jessica = new PersonCl('Jessica Davis', 1996);
jessica.calcAge();     // Method
console.log(jessica.age); // Getter (no parentheses)

PersonCl.hey(); // Static method
```

Class Rules

1. Classes are **NOT hoisted**
2. Classes are **first-class citizens** (can be passed/returned)
3. Classes always run in **strict mode**

📌 Getters and Setters

Work on any object:

```
const account = {
  movements: [200, 530, 120],

  get latest() {
    return this.movements.slice(-1).pop();
  },

  set latest(mov) {
    this.movements.push(mov);
  }
};

console.log(account.latest);  // 120 (getter)
account.latest = 50;          // setter
console.log(account.movements); // [200, 530, 120, 50]
```

📌 Object.create

Create object with specified prototype:

```
const PersonProto = {
  calcAge() {
    console.log(2037 - this.birthYear);
  },

  init(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  }
};

const steven = Object.create(PersonProto);
steven.init('Steven', 2002);
steven.calcAge();

console.log(steven.__proto__ === PersonProto); // true
```

📌 Inheritance: Constructor Functions

```
const Person = function(firstName, birthYear) {
  this.firstName = firstName;
  this.birthYear = birthYear;
};

Person.prototype.calcAge = function() {
  console.log(2037 - this.birthYear);
};

const Student = function(firstName, birthYear, course) {
  // Call parent constructor
  Person.call(this, firstName, birthYear);
  this.course = course;
};

// Link prototypes (BEFORE adding methods!)
Student.prototype = Object.create(Person.prototype);

// Fix constructor pointer
Student.prototype.constructor = Student;

// Add child methods
Student.prototype.introduce = function() {
  console.log(`I'm ${this.firstName} and I study ${this.course}`);
};
```

```
const mike = new Student('Mike', 2020, 'CS');
mike.introduce(); // Own method
mike.calcAge();   // Inherited method
```

📌 Inheritance: ES6 Classes

```
class PersonCl {
  constructor(fullName, birthYear) {
    this.fullName = fullName;
    this.birthYear = birthYear;
  }

  calcAge() {
    console.log(2037 - this.birthYear);
  }
}

class StudentCl extends PersonCl {
  constructor(fullName, birthYear, course) {
    // Must call super() first!
    super(fullName, birthYear);
    this.course = course;
  }

  introduce() {
    console.log(`I'm ${this.fullName} and I study ${this.course}`);
  }

  // Override parent method (polymorphism)
  calcAge() {
    console.log(`I'm ${2037 - this.birthYear} but feel older`);
  }
}

const martha = new StudentCl('Martha Jones', 2012, 'CS');
martha.introduce();
martha.calcAge(); // Uses overridden version
```

📌 Inheritance: Object.create

```
const PersonProto = {
  calcAge() {
    console.log(2037 - this.birthYear);
  },
  init(firstName, birthYear) {
    this.firstName = firstName;
    this.birthYear = birthYear;
  }
};

const StudentProto = Object.create(PersonProto);

StudentProto.init = function(firstName, birthYear, course) {
  PersonProto.init.call(this, firstName, birthYear);
  this.course = course;
};

const jay = Object.create(StudentProto);
jay.init('Jay', 2010, 'CS');
```

📌 Encapsulation: Private Fields (ES2022)

```
class Account {
  // Public fields
  locale = navigator.language;

  // Private fields (prefix with #)
  #movements = [];
  #pin;

  constructor(owner, currency, pin) {
    this.owner = owner;
    this.currency = currency;
    this.#pin = pin;
  }

  // Public API
  getMovements() {
    return this.#movements;
  }

  deposit(val) {
    this.#movements.push(val);
    return this;  // Enable chaining
  }

  withdraw(val) {
    this.deposit(-val);
    return this;
  }

  // Private method
  #approveLoan(val) {
    return true;
  }

  requestLoan(val) {
    if (this.#approveLoan(val)) {
      this.deposit(val);
    }
    return this;
  }
}

const acc = new Account('Jonas', 'EUR', 1111);

acc.deposit(250);
console.log(acc.#movements); // ❌ SyntaxError!
console.log(acc.getMovements()); // ✅ [250]
```

📌 Chaining Methods

Return `this` from methods:

```
acc.deposit(300)
   .withdraw(100)
   .requestLoan(25000)
   .withdraw(4000);
```

📌 Static Methods

Attached to class, not instances:

```
class Person {
  static hey() {
    console.log('Hey there!');
  }
}

Person.hey();     // ✅ Works
jonas.hey();      // ❌ Error

// Constructor function equivalent
Person.hey = function() {
  console.log('Hey there!');
};
```

🧪 Quick Reference Cheatsheet

```javascript
// Constructor function
const Person = function(name) {
  this.name = name;
};
Person.prototype.greet = function() {};

// ES6 Class
class Person {
  constructor(name) {
    this.name = name;
  }
  greet() {}           // Instance method
  get age() {}         // Getter
  set age(val) {}      // Setter
  static hey() {}      // Static method
  #privateField;       // Private field
  #privateMethod() {} // Private method
}

// Inheritance
class Student extends Person {
  constructor(name, course) {
    super(name);       // Must be first!
    this.course = course;
  }
}

// Object.create
const child = Object.create(parentProto);

// Check inheritance
obj instanceof Class;
obj.hasOwnProperty('prop');
```

✅ Exam Tips

1. **Never add methods** inside constructor (use prototype)
2. `new` creates object, sets `this`, links prototype, returns object
3. **Classes are NOT hoisted** (unlike function declarations)
4. `extends` + `super()` for class inheritance
5. **`super()` must be called first** in child constructor
6. Prefix private fields with `#`
7. Private fields must be **declared outside constructor**
8. Return `this` from methods to enable **chaining**
9. `Object.create(null)` creates object with no prototype
10. Use **Object.create** for pure prototypal inheritance
11. Child methods with same name **override** parent (polymorphism)
12. `static` methods are called on **class**, not instances