Aula 4

Created	@August 14, 2025 11:24 AM
Data	@August 14, 2025
Status	Iniciado
Tipo	Aula

Aula 4 – Herança, Polimorfismo, Atributos em Subclasses e `static`

> Módulo: FE-JS-003 – Programação Orientada a Objetos (JavaScript)
>
> Pré-requisitos: Classes, construtor, atributos, métodos, encapsulamento, getters/setters

---

0) Objetivos de aprendizagem

Ao final da aula, a turma será capaz de:

- Explicar como subclasses acessam atributos e métodos da classe base em JS.
- Diferenciar **público**, **privado (#)** e o padrão "**protegido**" (simulado) em JS.
- Usar `super` corretamente no construtor e em métodos sobrescritos.
- Aplicar **campos e métodos** `static` para contadores, registries, fábricas, caches e utilitários.
- Implementar **herança** e **polimorfismo** com exemplos práticos (catálogo/cardápio digital).

Perguntas rápidas

1. O que acontece se tentarmos acessar um **campo privado** (`#`) da classe pai a partir de uma subclasse?

2. Em que momento eu **preciso** chamar `super(...)`?
3. Em que casos um `static` faz mais sentido do que um campo de instância?

Mini-experimento:

```
class A {
  #segredo = 42;
  get segredo() {
    return this.#segredo;
  }
}
class B extends A {
  revelar() {
   /* return this.#segredo */
   }
}
const b = new B();
console.log(b.segredo); // 42
// b.revelar() quebraria se você tentasse acessar #segredo diretamente
```

---

2) Atributos em subclasses: público, privado e o "protegido"

2.1 Público

- Atributos criados como propriedades normais (ex.: `this.nome`) **são acessíveis** em subclasses via `this.nome`.

```
class Produto {
 constructor(nome) { this.nome = nome; }
}
class ProdutoFisico extends Produto {
 etiqueta() { return `Produto: ${this.nome}`; }
}
console.log(new ProdutoFisico('Mochila').etiqueta());
```

2.2 Privado com `#`

- Campos com `#` são **verdadeiramente privados** ao **corpo da classe onde foram declarados**. Subclasses **não** acessam.
- Use **getters/setters** ou métodos para expor comportamento, não o dado bruto.

```
class Conta {
  #saldo = 0;
  depositar(v){ this.#saldo += v; }
  get saldo(){ return this.#saldo; }
}
class ContaPremium extends Conta {
  bonus(){ this.depositar(10); }
  // this.#saldo → ERRO: não acessível
}
```

2.3 “Protegido” (padrão)

- JS **não** tem `protected`. Padrão comum:
  - Use **underscore** (`_interno`) para indicar intenção de “não público”.
  - Ou exponha **getters**/métodos específicos para que a subclasse use, mantendo invariantes.

```
class Autenticavel {
  constructor(){ this._roles = new Set(); }
  hasRole(r){ return this._roles.has(r); }
}
class Usuario extends Autenticavel {
  concederRole(r){ this._roles.add(r); } // acesso “protegido” por convenção
}
```

2.4 Ordem e regras de `super`

- Em **subclasses**, **chame** `super(...)` **antes** de usar `this` no construtor.
- `super.metodo()` permite reutilizar lógica da classe base em métodos sobrescritos.

```
class Produto {
  constructor(nome, preco){ this.nome = nome; this.preco = preco; }
  precoFinal(){ return this.preco; }
}
class ProdutoFisico extends Produto {
  constructor(nome, preco, embalagem){
    super(nome, preco); // obrigatório antes de usar this
    this.embalagem = embalagem ?? 0;
  }
  precoFinal(){ return super.precoFinal() + this.embalagem; }
}
```

> Pergunta comum: “Por que em uma classe passo params via super e em outra não?”
>
> **Resposta:** se a subclasse **define** `constructor`, você precisa **delegar** para o construtor da base com os parâmetros necessários; se **não define**, o construtor padrão chama `super(...args)` para você.

---

3) `static`: campos e métodos da classe

3.1 Conceito

- **Pertencem à classe**, não à instância. Acessamos via `MinhaClasse.algo`, não via `obj.algo`.
- Úteis para **constantes**, **contadores globais**, **fábricas**, **validadores**, **registries** e **caches**.
- `this` dentro de um método `static` referencia a **própria classe** (útil em heranças).

3.2 Exemplos práticos

Contador de instâncias & ID incremental

```
class Pedido {
  static #seq = 0;            // privado e estático
  static gerarId(){ return ++this.#seq; } // usa `this` para herdar bem
  constructor(){ this.id = Pedido.gerarId(); }
}
console.log(new Pedido().id, new Pedido().id); // 1 2
```

Fábrica (factory) e validador

```
class Produto {
  constructor(nome, preco){ this.nome = nome; this.preco = preco; }
  static fromJSON(json){ const {nome, preco} = JSON.parse(json); return n
ew this(nome, preco); }
  static isProduto(x){ return x instanceof Produto; }
}
const p = Produto.fromJSON('{"nome":"Café","preco":12.5}');
console.log(Produto.isProduto(p)); // true
```

Registry/Catálogo compartilhado

```
class Catalogo {
  static itens = new Map(); // compartilhado entre todas as instâncias
  static cadastrar(prod){ this.itens.set(prod.nome, prod); }
  static buscar(nome){ return this.itens.get(nome); }
}
```

3.3 `static` em herança

- **Métodos/fields estáticos são herdados**.
- `super` pode ser usado em contexto estático para reusar fábricas/validadores.

```
class Base { static tipo = 'base'; static quemSou(){ return this.tipo; } }
class Derivada extends Base { static tipo = 'derivada'; }
console.log(Base.quemSou());     // 'base'
console.log(Derivada.quemSou()); // 'derivada' (polimorfismo estático via `t
his`)
```

Blocos estáticos (inicialização complexa)

```
class Env {
  static config = {};
  static {
    // carregar valores padrões/derivados
    this.config.API_URL = 'https://api.exemplo.com';
  }
}
```

> Boas práticas: evite usar static para estado mutável que deveria ser de cada instância. Prefira-o para metadados, utilitários e caches controlados.

To-Do List OO (CRUD completo)

> Problema gerador: Como criar uma To-Do List do zero aplicando POO em JavaScript (classes, encapsulamento, herança/polimorfismo quando fizer sentido), CRUD completo, persistência e organização por camadas?

Modelagem

Entidade principal

- **Todo**: `id`, `title`, `done`, `createdAt`
- Regras:
  - `title` é **obrigatório** (trim), máx. 120 chars.
  - `done` é booleano; alterna com `toggle()`.
  - `id` é **gerenciado** via `static nextId`.

Casos de uso (use cases)

- Criar tarefa
- Listar tarefas (todas / ativas / concluídas)
- Editar título
- Alternar concluída
- Excluir
- (Extra) Limpar concluídas, estatísticas

Estrutura de pastas (ES Modules)

```
/to-do-oo
  ├─ index.html
  ├─ edit.html
  ├─ styles.css
  └─ src/
    ├─ model/Todo.js
    ├─ infra/TodoRepository.js
    ├─ domain/TodoService.js
    ├─ ui/Controller.js
    ├─ pages/edit.js
    └─ main.js
```

> Nota didática – Por que instâncias e não tudo static?
>
> Métodos/atributos `static` pertencem à **classe** (1 só estado global), não à **instância**. Para regras que operam sobre **dados específicos de cada tarefa** (ex.: `title`, `done`), precisamos de instâncias. O `static` é útil para **utilidades** (ex.: `fromJSON`) e **metadados globais** (ex.: `nextId`, `STORAGE_KEY`).

Implementação (passo a passo)

1) `index.html`

```html
<!doctype html>
<html lang="pt-br">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>To-Do List OO – ADA</title>
  <link rel="stylesheet" href="./styles.css" />
</head>
<body>
  <main class="container">
    <h1>To-Do List (POO)</h1>

    <form id="new-form">
      <input id="new-title" type="text" placeholder="Nova tarefa..." autocom
plete="off" required />
      <button type="submit">Adicionar</button>
    </form>

    <div id="filters" class="filters">
      <button data-filter="all" class="active">Todas</button>
      <button data-filter="active">Ativas</button>
      <button data-filter="completed">Concluídas</button>
    </div>

    <p id="counters" class="counters">Total: 0 | Ativos: 0 | Concluídos: 0</
p>

    <ul id="todo-list" class="list"></ul>
  </main>

  <script type="module" src="./src/main.js"></script>
</body>
</html>
```

2) edit.html

```html
<!doctype html>
<html lang="pt-br">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Editar tarefa – To-Do OO</title>
  <link rel="stylesheet" href="./styles.css" />
</head>
<body>
  <main class="container">
    <h1>Editar tarefa</h1>
    <form id="edit-form">
      <input id="title" type="text" required />
      <button type="submit">Salvar</button>
      <a class="link" href="./index.html">Cancelar</a>
    </form>
  </main>

  <script type="module" src="./src/pages/edit.js"></script>
</body>
</html>
```

3) `styles.css` (mínimo para visualização)

```css
* { box-sizing: border-box; }
body { font-family: system-ui, sans-serif; margin: 0; padding: 24px; backgr
ound: #0d1117; color: #c9d1d9; }
.container { max-width: 720px; margin: 0 auto; }
form { display: flex; gap: 8px; margin: 16px 0; }
input[type="text"] { flex: 1; padding: 10px 12px; border: 1px solid #30363d;
background: #161b22; color: #c9d1d9; border-radius: 6px; }
button { padding: 10px 12px; border: 1px solid #30363d; background: #238
636; color: #fff; border-radius: 6px; cursor: pointer; }
button.active { outline: 2px solid #58a6ff; }
.filters { display: flex; gap: 8px; }
.list { list-style: none; padding: 0; margin: 16px 0; display: grid; gap: 8px; }
.list li { display: grid; grid-template-columns: 24px 1fr auto auto; align-item
```

```
s: center; gap: 8px; padding: 8px; border: 1px solid #30363d; border-radiu
s: 6px; background: #161b22; }
.title.done { text-decoration: line-through; opacity: .7; }
.link { margin-left: 8px; color: #58a6ff; text-decoration: none; }
```

4) src/model/Todo.js

```
// src/model/Todo.js
export class Todo {
  static nextId = 1; // gerenciado pelo repositório ao carregar

  #id; #title; #done; #createdAt;

  constructor({ id, title, done = false, createdAt = new Date().toISOString()
}) {
    this.#id = id ?? Todo.nextId++;
    this.title = title; // valida via setter
    this.#done = Boolean(done);
    this.#createdAt = createdAt;
  }

  get id() { return this.#id; }
  get title() { return this.#title; }
  set title(value) {
    const v = String(value ?? '').trim();
    if (!v) throw new Error('Título é obrigatório');
    if (v.length > 120) throw new Error('Título deve ter no máximo 120 caract
eres');
    this.#title = v;
  }
  get done() { return this.#done; }
  toggle() { this.#done = !this.#done; }
  get createdAt() { return this.#createdAt; }

  toJSON() { return { id: this.#id, title: this.#title, done: this.#done, createdA
t: this.#createdAt }; }
```

```
  static fromJSON(json) { return new Todo(json); }
}
```

5) src/infra/TodoRepository.js

```
// src/infra/TodoRepository.js
import { Todo } from '../model/Todo.js';

const STORAGE_KEY = 'ada.todo.v1';

export class TodoRepository {
  static STORAGE_KEY = STORAGE_KEY; // exemplo de uso de static para
metadado global

  constructor(storage = window.localStorage) {
    this.storage = storage;
    this.#ensureInit();
  }

  #ensureInit() {
    if (!this.storage.getItem(STORAGE_KEY)) {
      this.storage.setItem(STORAGE_KEY, JSON.stringify([]));
    }
    // Ajusta nextId com base no maior id existente (persistência)
    const items = JSON.parse(this.storage.getItem(STORAGE_KEY) || '[]');
    const maxId = items.reduce((m, t) => Math.max(m, Number(t.id) || 0), 0);
    Todo.nextId = Math.max(Todo.nextId, maxId + 1);
  }

  #readAll() { return JSON.parse(this.storage.getItem(STORAGE_KEY) ||
'[]'); }
  #writeAll(items) { this.storage.setItem(STORAGE_KEY, JSON.stringify(item
s)); }

  findAll() { return this.#readAll().map(Todo.fromJSON); }
  findById(id) {
    const it = this.#readAll().find(t => Number(t.id) === Number(id));
    return it ? Todo.fromJSON(it) : null;
```

```
  }
  save(todo) {
    const items = this.#readAll();
    items.push(todo.toJSON());
    this.#writeAll(items);
    return todo;
  }
  update(todo) {
    const items = this.#readAll();
    const idx = items.findIndex(t => Number(t.id) === Number(todo.id));
    if (idx === -1) return null;
    items[idx] = todo.toJSON();
    this.#writeAll(items);
    return todo;
  }
  delete(id) {
    const items = this.#readAll();
    const newItems = items.filter(t => Number(t.id) !== Number(id));
    this.#writeAll(newItems);
    return newItems.length !== items.length;
  }
  clearCompleted() {
    const items = this.#readAll();
    const newItems = items.filter(t => !t.done);
    this.#writeAll(newItems);
  }
}
```

6) src/domain/TodoService.js

```
// src/domain/TodoService.js
import { Todo } from '../model/Todo.js';
import { TodoRepository } from '../infra/TodoRepository.js';

export class TodoService {
  constructor(repo = new TodoRepository()) { this.repo = repo; }

  create(title) { const todo = new Todo({ title }); return this.repo.save(todo);
```

```
  }

  edit(id, newTitle) {
    const todo = this.repo.findById(id);
    if (!todo) throw new Error('Todo não encontrado');
    todo.title = newTitle; // valida via setter
    return this.repo.update(todo);
  }

  toggle(id) {
    const todo = this.repo.findById(id);
    if (!todo) throw new Error('Todo não encontrado');
    todo.toggle();
    return this.repo.update(todo);
  }

  remove(id) { return this.repo.delete(id); }

  list(filter = 'all') {
    const all = this.repo.findAll();
    if (filter === 'active') return all.filter(t => !t.done);
    if (filter === 'completed') return all.filter(t => t.done);
    return all;
  }

  stats() {
    const all = this.repo.findAll();
    const completed = all.filter(t => t.done).length;
    return { total: all.length, completed, active: all.length - completed };
  }

  getById(id) { return this.repo.findById(id); }
}
```

7) src/ui/Controller.js

```
// src/ui/Controller.js
import { TodoService } from '../domain/TodoService.js';
```

```
export class Controller {
  constructor(doc = document, service = new TodoService()) {
    this.doc = doc; this.service = service;
    this.form = doc.querySelector('#new-form');
    this.input = doc.querySelector('#new-title');
    this.list = doc.querySelector('#todo-list');
    this.counters = doc.querySelector('#counters');
    this.filters = doc.querySelector('#filters');

    this.currentFilter = 'all';
    this.#bind();
    this.render();
  }

  #bind() {
    this.form.addEventListener('submit', (e) => {
      e.preventDefault();
      try {
        this.service.create(this.input.value);
        this.input.value = '';
        this.render();
      } catch (err) { alert(err.message); }
    });

    this.list.addEventListener('click', (e) => {
      const li = e.target.closest('li[data-id]');
      if (!li) return;
      const id = Number(li.dataset.id);

      if (e.target.matches('.toggle')) { this.service.toggle(id); this.render(); }
      if (e.target.matches('.delete')) { this.service.remove(id); this.render(); }
      if (e.target.matches('.edit')) { window.location.href = `edit.html?id=${id}
`; }
    });

    this.filters.addEventListener('click', (e) => {
      if (e.target.matches('button[data-filter]')) {
```

```
        this.currentFilter = e.target.dataset.filter;
        this.render();
      }
    });
  }

  render() {
    const todos = this.service.list(this.currentFilter);
    this.list.innerHTML = todos.map(t => `
      <li data-id="${t.id}">
        <input type="checkbox" class="toggle" ${t.done ? 'checked' : ''} />
        <span class="title ${t.done ? 'done' : ''}">${t.title}</span>
        <button class="edit">Editar</button>
        <button class="delete">Excluir</button>
      </li>
    `).join('');

    const s = this.service.stats();
    this.counters.textContent = `Total: ${s.total} | Ativos: ${s.active} | Conclu
ídos: ${s.completed}`;

    this.doc.querySelectorAll('#filters button')
      .forEach(b => b.classList.toggle('active', b.dataset.filter === this.current
Filter));
  }
}
```

8) src/pages/edit.js

```
// src/pages/edit.js
import { TodoService } from '../domain/TodoService.js';

const params = new URLSearchParams(location.search);
const id = Number(params.get('id'));
const service = new TodoService();

const input = document.querySelector('#title');
const form = document.querySelector('#edit-form');
```

```
(function init() {
  const todo = service.getById(id);
  if (!todo) {
    alert('Tarefa não encontrada');
    location.href = 'index.html';
    return;
  }
  input.value = todo.title;
})();

form.addEventListener('submit', (e) => {
  e.preventDefault();
  try {
    service.edit(id, input.value);
    location.href = 'index.html';
  } catch (err) { alert(err.message); }
});
```

9) `src/main.js`

```
// src/main.js
import { Controller } from './ui/Controller.js';
new Controller();
```

Atividade de Fixação 4 (escopo fechado)

1. **Adicionar um botão “Editar”** (já presente na implementação).
2. **Criar uma página de edição** com os campos **preenchidos** (arquivo `edit.html` + `src/pages/edit.js`).
3. **Salvar e voltar** para a tela principal após editar.

Critérios de aceite

- Editar um item existente **mantém o** `id` **e** `done`, alterando apenas `title` (com validação).
- Voltar para `index.html` após salvar com a lista atualizada.

Desafios extras (para quem terminar antes / tarefa de casa)

- **Limpar concluídas** (botão que chama `repo.clearCompleted()` via Service).
- **Ordenação** (recentes primeiro / concluídas no fim).
- **Filtro por texto** (campo que filtra por substring no título).
- **Barra de progresso** (concluídas ÷ total).
- **Keyboard UX** (Enter para salvar, Esc para cancelar na edição).
- **Testes** (Jest) para `TodoService` com repositório “in-memory”.