

AUTENTICACÃO  
E AUTORIZACÃO

Quando desenvolvemos uma aplicação web, garantir que apenas usuários permitidos tenham acesso a certas informações ou funcionalidades é essencial. É aí que entram dois conceitos muito importantes: **autenticação e autorização.**

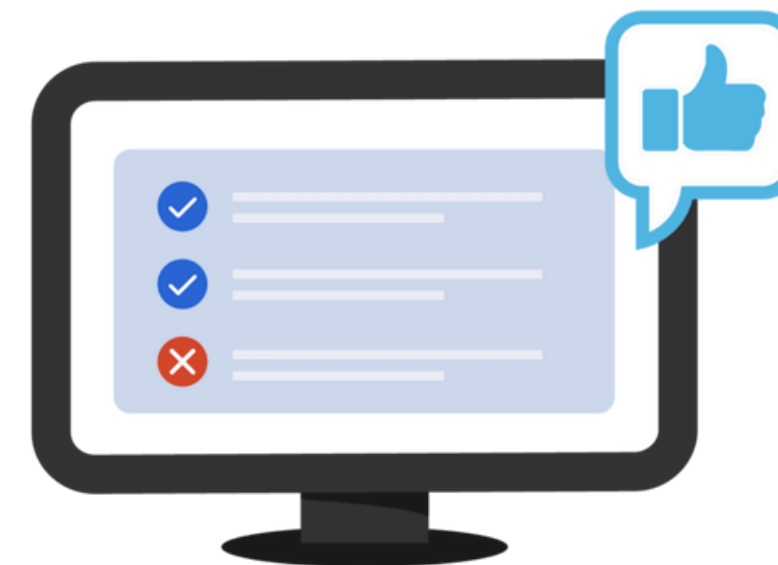
- **Autenticação (Authentication)** é o processo de **verificar quem é o usuário**. Exemplo clássico: login com email e senha.
- **Autorização (Authorization)** é o processo de **verificar o que esse usuário pode fazer**. Por exemplo, um usuário logado pode acessar o painel de controle, mas não alterar dados de outros usuários se não tiver permissão.

### Authentication



Confirms users  
are who they say they are.

### Authorization

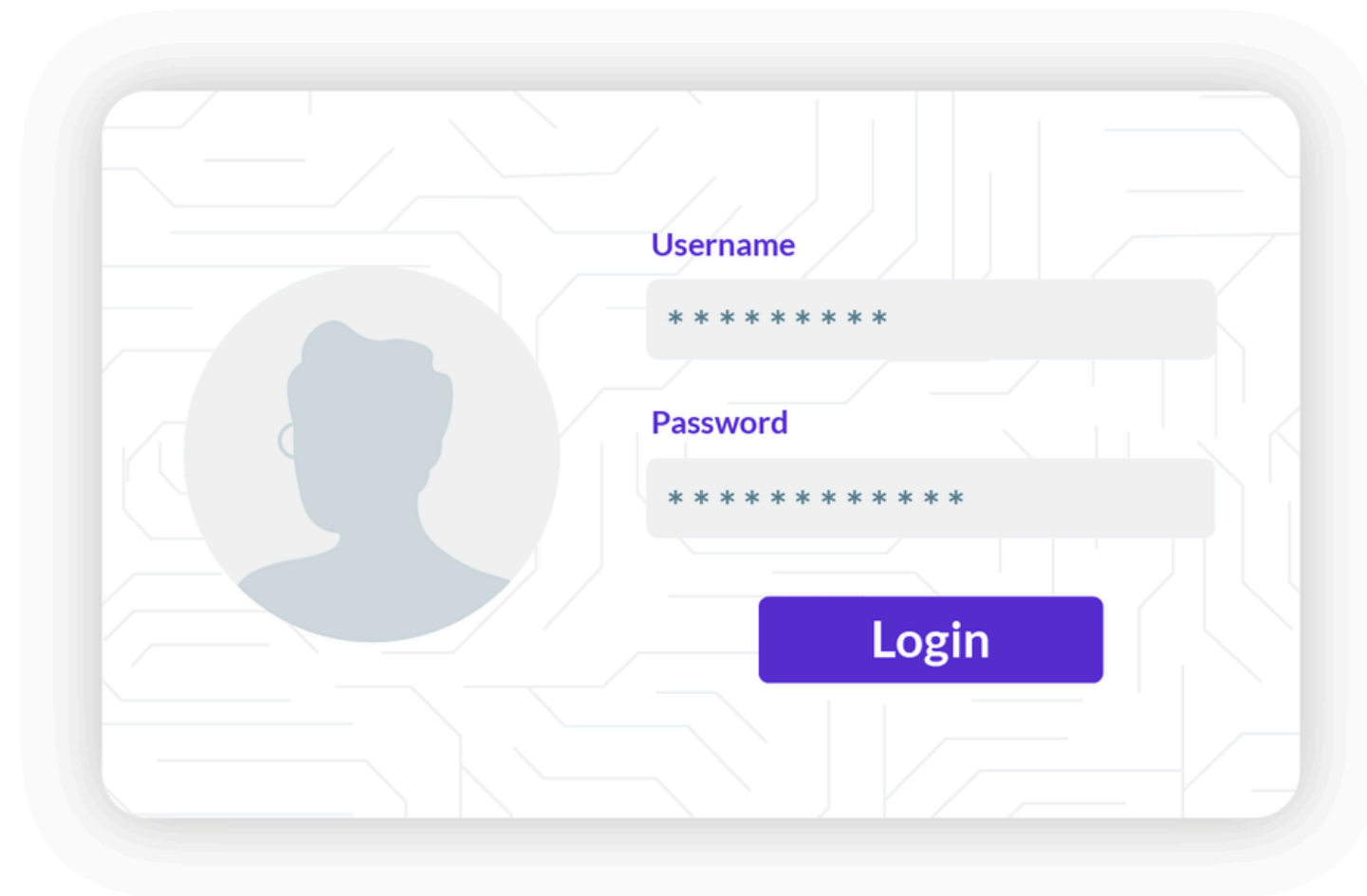


Gives users permission  
to access a resource.

Autenticação responde “Quem é você?”

Autorização responde “O que você pode fazer?”

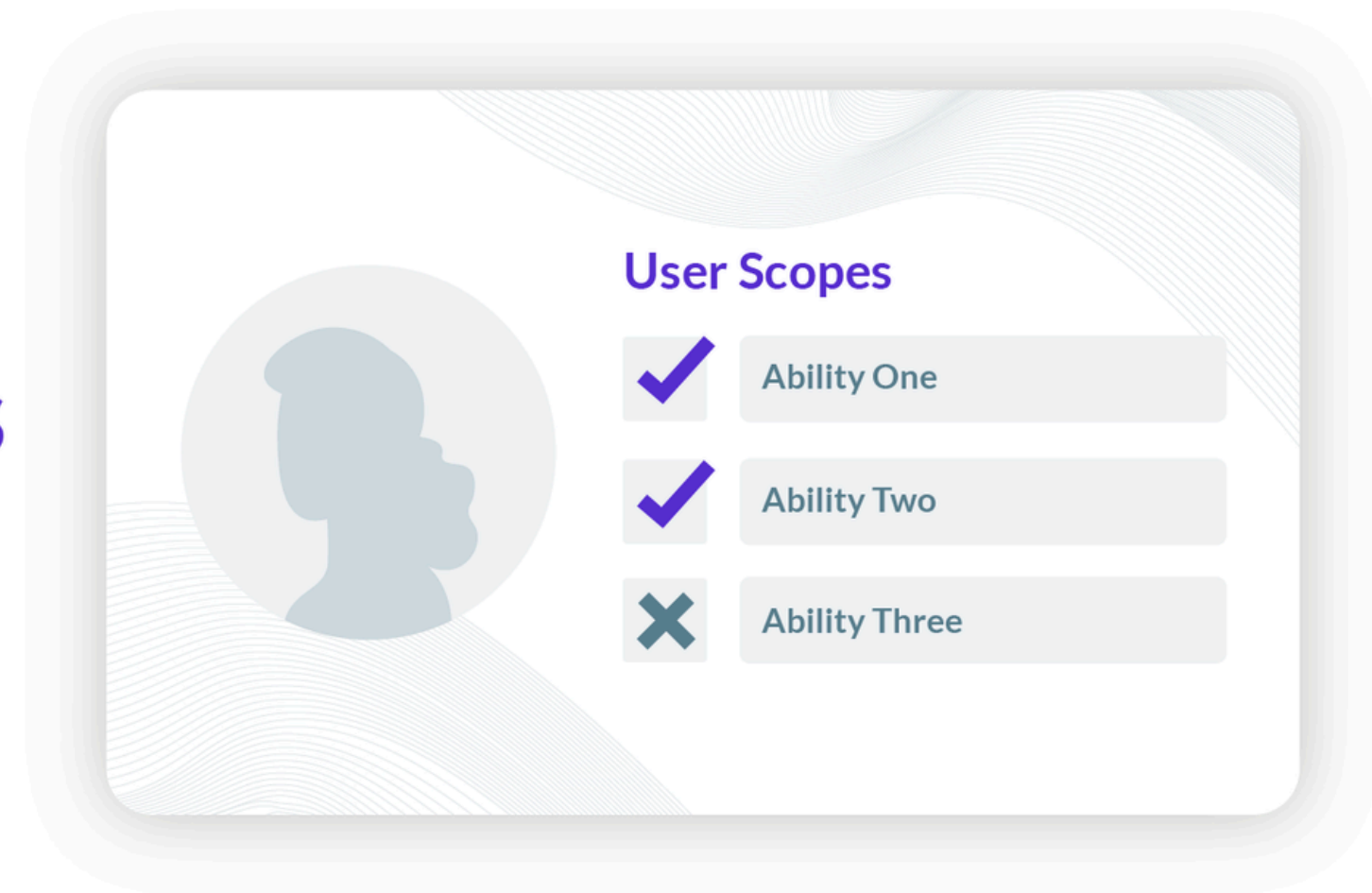
## AUTHENTICATION



A diagram of an authentication interface. It features a light gray rounded rectangle with a background of white circuit-like lines. On the left is a circular profile icon of a person. To the right of the icon are two input fields: the top one is labeled 'Username' and contains ten asterisks; the bottom one is labeled 'Password' and contains sixteen asterisks. Below these fields is a solid blue button with the word 'Login' in white text.

VS

## AUTHORIZATION



A diagram of an authorization interface. It features a light gray rounded rectangle with a background of white wavy lines. On the left is a circular profile icon of a person. To the right of the icon is the title 'User Scopes' in blue. Below the title are three items, each in a light gray box. The first two items have a blue checkmark icon to their left and are labeled 'Ability One' and 'Ability Two'. The third item has a green 'X' icon to its left and is labeled 'Ability Three'.

O QUE É WWT?

**JWT (JSON Web Token)** é um padrão para transmitir dados de forma segura entre duas partes. É um **token** (tipo um “cartão de acesso digital”) assinado que contém informações do usuário — como seu ID e email — e pode ser verificado sem necessidade de guardar nada no servidor.

# COMO FUNCIONA A AUTENTICAÇÃO COM LOGIN, SENHA E JWT?

Vamos ver o **fluxo básico** em 5 passos:

## ➡ **Passo 1: Usuário envia login e senha**

- Ele preenche um formulário.
- Você recebe esses dados no backend e valida com o banco de dados.



```
1 // Exemplo com Express + Prisma
2 const user = await prisma.user.findUnique({ where: { email } });
3 const passwordMatch = await bcrypt.compare(password, user.password);
```

# COMO FUNCIONA A AUTENTICAÇÃO COM LOGIN, SENHA E JWT?

## 🔑 Passo 2: Geração do token JWT

- Se o login estiver certo, você **gera um token JWT** com informações básicas do usuário (sem dados sensíveis!).

```
1 import jwt from 'jsonwebtoken';  
2  
3 const token = jwt.sign(  
4   { id: user.id, email: user.email }, // payload  
5   process.env.JWT_SECRET,           // chave secreta  
6   { expiresIn: '1h' }                // tempo de expiração  
7 );
```



# COMO FUNCIONA A AUTENTICAÇÃO COM LOGIN, SENHA E JWT?

## 📦 Passo 3: Envio do token pro cliente

- O token pode ser enviado no **body**, **cookie** ou **header** da resposta. O mais comum em SPA é via *Authorization Header*:

```
1 {  
2   "token": "eyJhbGciOiJIUzI1NiIsInR..."  
3 }
```

# COMO FUNCIONA A AUTENTICAÇÃO COM LOGIN, SENHA E JWT?

## 📡 Passo 4: Cliente envia o token nas próximas requisições

- Toda vez que o cliente fizer uma requisição protegida, ele envia o token no header:



```
1 Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR...
```

# COMO FUNCIONA A AUTENTICAÇÃO COM LOGIN, SENHA E JWT?

## ✓ Passo 5: Backend valida o token

- No backend, você usa o `jwt.verify()` para **validar a assinatura do token** e recuperar o *payload*.



```
1 const decoded = jwt.verify(token, process.env.JWT_SECRET);  
2 // decoded → { id: ..., email: ..., iat: ..., exp: ... }
```

# AUTENTICAÇÃO EM 2 FATORES (2FA)

É uma camada extra de segurança: além da **senha (1º fator)**, o usuário precisa fornecer um **segundo fator** para confirmar sua identidade.

## Tipos comuns:

- **Token temporário (TOTP):** via apps como Google Authenticator.
- **Código por SMS ou email.**
- **Chave física (como YubiKey).**

## Exemplo de fluxo:

1. Usuário faz login com email e senha.
2. Backend envia um código (via email, SMS ou app).
3. Usuário digita o código → backend valida.

## Como implementar:

- Libs: speakeasy (para gerar/verificar códigos TOTP), qrcode (para gerar QR Codes).
- Usa o algoritmo TOTP com chave secreta salva para o usuário.

# AUTENTICAÇÃO BIONMÉTRICA

Usa características físicas para autenticar: impressão digital, reconhecimento facial, etc.

## Onde é usada:

- **Apps mobile:** usando APIs nativas (ex: react-native-fingerprint-scanner, Expo LocalAuthentication).
- **Web:** via **WebAuthn** (nova API de autenticação da Web).

## Exemplo no navegador:

- **navigator.credentials.get()** com WebAuthn.
- Permite login com digital, Face ID ou chave de segurança.

# COMPARANDO COM LOGIN TRADICIONAL

Usa características físicas para autenticar: impressão digital, reconhecimento facial, etc.

Método	Segurança	Facilidade	Recomendado para
Senha	Média	Alta	Todos os apps
2FA (código)	Alta	Média	Apps sensíveis
Biométrica/WebAuthn	Muito Alta	Alta	Mobile e apps críticos

HERANCA

SUBCLASSE E SUPERCLASSE

# HERANÇA

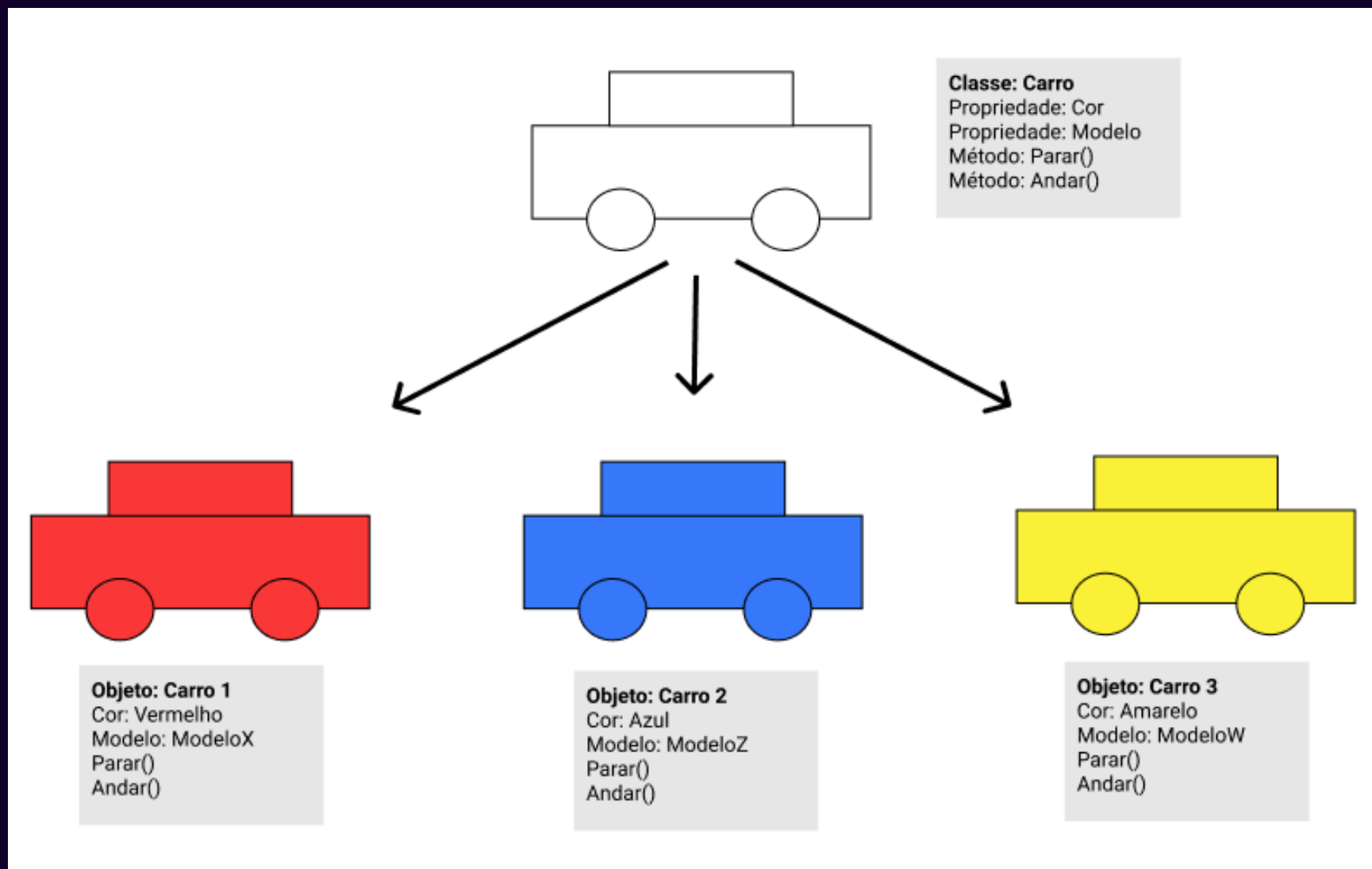
Herança é um conceito fundamental em orientação a objetos que permite que uma classe (chamada de classe filha ou subclasse) herde propriedades e métodos de outra classe (chamada de classe pai ou superclasse). Isso permite a criação de uma nova classe que é uma extensão da classe existente, reutilizando e especializando funcionalidades.

**SABER MAIS**



# EXEMPLO ABSTRATO: CARROS

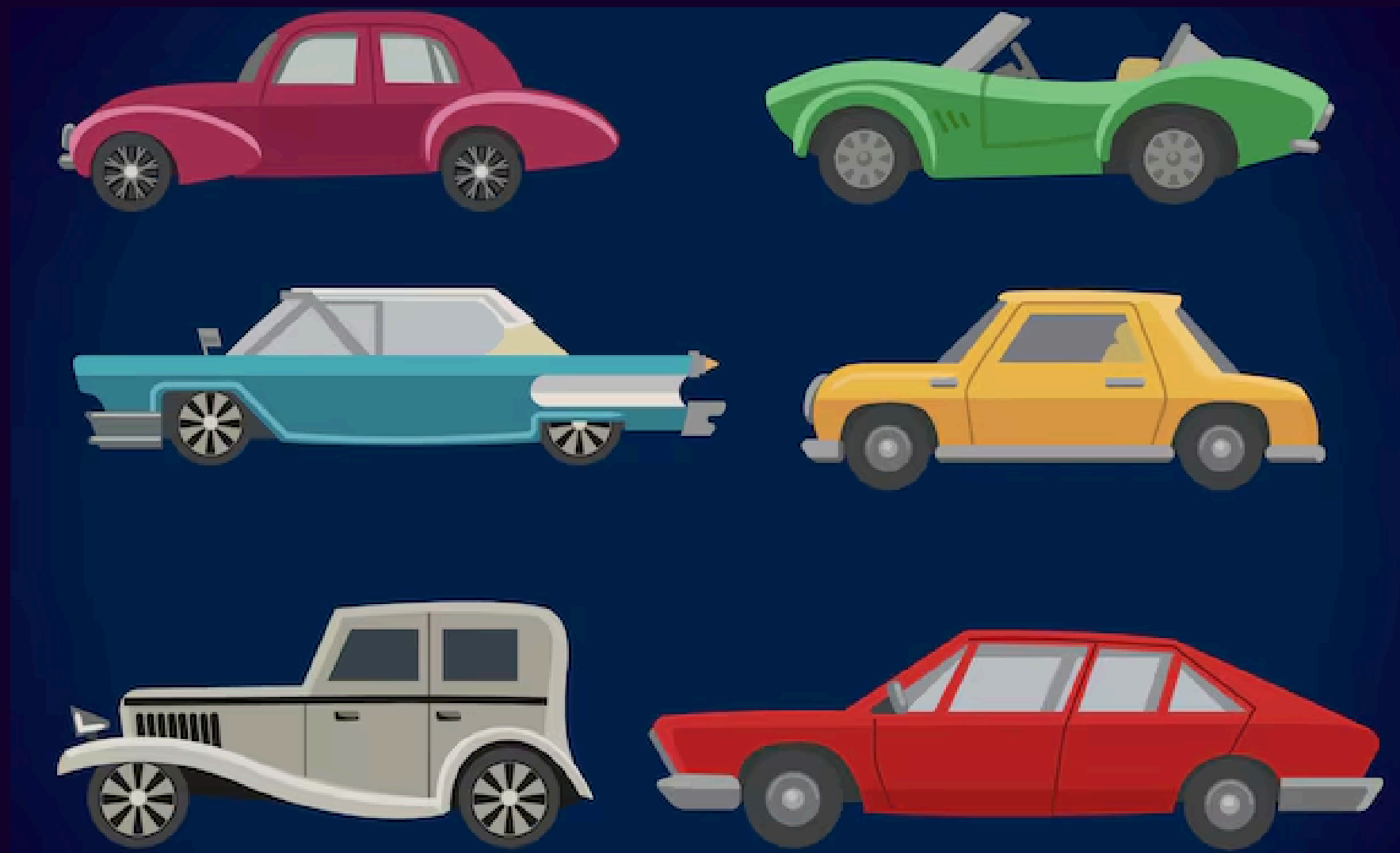
Imagine que temos uma classe base chamada **Carro**. Esta classe contém atributos e métodos comuns a todos os carros.



```
1 class Carro {
2   constructor(marca, modelo, ano) {
3     this.marca = marca;
4     this.modelo = modelo;
5     this.ano = ano;
6   }
7
8   andar() {
9     console.log(`${this.marca} ${this.modelo} do ano ${this.ano} está andando.`);
10  }
11
12  parar() {
13    console.log(`${this.marca} ${this.modelo} do ano ${this.ano} está parando.`);
14  }
15 }
16
17 // Instanciando objetos da classe Carro
18 const carro1 = new Carro('Toyota', 'Corolla', 2020);
19 const carro2 = new Carro('Honda', 'Civic', 2018);
20 const carro3 = new Carro('Chevrolet', 'Onix', 2021);
21
22 // Utilizando os métodos
23 carro1.andar(); // "Toyota Corolla do ano 2020 está andando."
24 carro1.parar(); // "Toyota Corolla do ano 2020 está parando."
25
26 carro2.andar(); // "Honda Civic do ano 2018 está andando."
27 carro2.parar(); // "Honda Civic do ano 2018 está parando."
28
29 carro3.andar(); // "Chevrolet Onix do ano 2021 está andando."
30 carro3.parar(); // "Chevrolet Onix do ano 2021 está parando."
```

# CLASSE DERIVADA: CARROESPORTIVO

Agora, vamos criar uma classe derivada chamada **CarroEsportivo**, que representa um tipo específico de carro, um carro esportivo. Esta classe herda as propriedades e métodos da classe **Carro**, mas também adiciona funcionalidades específicas para carros esportivos.



```
1 class CarroEsportivo extends Carro {
2     constructor(marca, modelo, ano, velocidadeMaxima) {
3         super(marca, modelo, ano); // Chama o construtor da classe Carro
4         this.velocidadeMaxima = velocidadeMaxima;
5     }
6
7     acelerar() {
8         console.log(`${this.marca} ${this.modelo} está acelerando a ${this.velocidadeMaxima} km/h.`);
9     }
10 }
```

Na classe **CarroEsportivo**, o método `super` é usado para chamar o construtor da classe **Carro** e inicializar os atributos herdados. Além disso, adicionamos um novo atributo **velocidadeMaxima** e um método `acelerar` específico para carros esportivos.

```
1 const meuCarro = new Carro('Toyota', 'Corolla', 2021);
2 meuCarro.ligar(); // Saída: Toyota Corolla está ligado.
3 meuCarro.desligar(); // Saída: Toyota Corolla está desligado.
4
5 const meuCarroEsportivo = new CarroEsportivo('Ferrari', '488', 2022, 340);
6 meuCarroEsportivo.ligar(); // Saída: Ferrari 488 está ligado.
7 meuCarroEsportivo.acelerar(); // Saída: Ferrari 488 está acelerando a 340 km/h.
```

O meuCarro usa os métodos ligar e desligar da classe Carro, enquanto o meuCarroEsportivo usa esses mesmos métodos, além do método acelerar adicionado pela classe CarroEsportivo.

# MODULARIZAÇÃO

A herança permite que a classe CarroEsportivo reutilize o código da classe Carro, evitando duplicação e facilitando a manutenção do código.

Carros esportivos têm todas as funcionalidades de um carro comum, mas também podem ter funcionalidades adicionais específicas para sua categoria. Essa abordagem torna o código mais modular e organizado, seguindo o princípio da reutilização e especialização.

[SABER MAIS](#)

# SUPER()

O método `super()` é uma parte essencial do sistema de herança em JavaScript, especialmente quando se trabalha com classes. Ele é utilizado para acessar e chamar métodos da classe pai (ou superclasse) a partir da classe filha (ou subclasse). Aqui está uma explicação detalhada sobre como e por que usar o `super()`:

```
1 class Animal {
2   constructor(nome) {
3     this.nome = nome;
4   }
5
6   fazerSom() {
7     console.log(`${this.nome} faz um som.`);
8   }
9 }
10
11 class Cachorro extends Animal {
12   constructor(nome, raca) {
13     super(nome); // Chama o construtor da superclasse Animal
14     this.raca = raca;
15   }
16
17   fazerSom() {
18     super.fazerSom(); // Chama o método da superclasse Animal
19     console.log(`${this.nome} faz au au.`);
20   }
21 }
22
23 const meuCachorro = new Cachorro('Rex', 'Labrador');
24 meuCachorro.fazerSom();
25 // Saída:
26 // Rex faz um som.
27 // Rex faz au au.
```



# SOBREESCREVENDO MÉTODOS NA CLASSE

01

A classe Dog sobrescreve o método fazerSom() da classe Animal. Quando a instância de Cachorro chama fazerSom(), a versão de Cachorro é executada em vez da versão da classe base. Isso permite personalizar o comportamento de métodos herdados. Se você quiser, pode usar o método da classe base dentro do método sobrescrito da classe estendida com a função super().

```
1 class Cachorro extends Animal {  
2   fazerSom() {  
3     super.fazerSom(); // Chama o método `fazerSom()` da classe `Animal`  
4     console.log(`${this.nome} também está latindo!`);  
5   }  
6 }
```

# ADICÃO DE NOVOS MÉTODOS NA CLASSE

02

Além de sobrescrever métodos, podemos adicionar novos métodos nas classes estendidas que não existem na classe base. Isso permite que as subclasses tenham funcionalidades adicionais que não estão disponíveis nas classes de onde herdam.

```
1 class Cachorro extends Animal {  
2     fazerSom() {  
3         super.fazerSom(); // Chama o método `fazerSom()` da classe `Animal`  
4         console.log(`${this.nome} também está latindo!`);  
5     }  
6  
7     brincar() {  
8         console.log(`${this.nome} está brincando com a bola.`);  
9     }  
10 }
```



# VANTAGENS

01

## **Reutilização de Código:**

Você pode criar funcionalidades comuns em uma classe base e reutilizá-las em várias subclasses, evitando duplicação de código.

02

## **Personalização:**

Subclasses podem sobrescrever métodos herdados para adaptar comportamentos sem modificar a classe base.

03

## **Expansão:**

Além de sobrescrever, você pode adicionar novos métodos à subclasse, expandindo suas capacidades.

# OBSERVAÇÕES

01

Classes **não** são "hoisted"

02

Uso obrigatório de `super()` nas subclasses

03

Classes tem strict mode ativado por padrão

04

Métodos em classes não podem ser usados antes da declaração

05

As classes podem ter métodos estáticos definidos

06

Em métodos de classe, o "this" sempre se refere à instância da classe

THANK  
YOU

@wallace027dev