

Téc em Desenvolvimento
de Sistemas Bilíngue

Desenvolver Código

Orientado a Objetos

Herança

- A Herança é um artifício que permite que criemos classes com características de outra
- Uma classe que herda propriedades e métodos de outra é chamada de subclass (subclasse) ou child class (classe filha)

Herança

- A classe que dá essas características é chamada de superclass (superclasse), base class (classe base) ou parent class (classe pai)
- As classes podem ser parent de outra classe e child de outra ao mesmo tempo (como na vida real)

```
export class Dog {

    name: string;
    weight: number;

    constructor(name: string, weight: number){
        this.name = name;
        this.weight = weight;
    }

    bark(): void {
        console.log("au au 🐶");
    }

    eat(quantity: number): void {
        console.log("the dog has eaten " + quantity);
    }

}
```

```
export class Owl {

    name: string;
    weight: number;

    constructor(name: string, weight: number){
        this.name = name;
        this.weight = weight;
    }

    chirp(): void {
        console.log("Hu Hu 🦉");
    }

    eat(quantity: number): void {
        console.log("the owl has eaten " + quantity);
    }

    fly(quantity: number): void {
        console.log("the owl has flown for " + quantity + " minutes");
    }

}
```

```
export class Dog {
```

```
  name: string;  
  weight: number;
```

```
  constructor(name: string, weight: number){  
    this.name = name;  
    this.weight = weight;  
  }
```

```
  bark(): void {  
    console.log("au au 🐶");  
  }
```

```
  eat(quantity: number): void {  
    console.log("the dog has eaten " + quantity);  
  }
```

```
}
```

```
export class Owl {
```

```
  name: string;  
  weight: number;
```

```
  constructor(name: string, weight: number){  
    this.name = name;  
    this.weight = weight;  
  }
```

```
  chirp(): void {  
    console.log("Hu Hu 🦉");  
  }
```

```
  eat(quantity: number): void {  
    console.log("the owl has eaten " + quantity);  
  }
```

```
  fly(quantity: number): void {  
    console.log("the owl has flown for " + quantity + " minutes");  
  }
```

```
}
```

Herança

- Nós podemos, e devemos, isolar tudo que for similar em uma classe pai. No exemplo anterior, podemos chama-la de Animal.

```
export class Animal {  
  
    name: string;  
    weight: number;  
  
    constructor(name: string, weight: number){  
        this.name = name;  
        this.weight = weight;  
    }  
  
    eat(quantity: number): void {  
        console.log("the animal has eaten " + quantity);  
    }  
  
}
```


Herança

- O próximo passo é transformar as classes em filhas da classe criada, para isso usaremos o `extends`


```
export class Dog extends Animal {  
    constructor(name: string, weight: number) {  
        super(name, weight);  
    }  
  
    bark(): void {  
        console.log("au au 🐕");  
    }  
}
```

Herança

- extends indica que a classe vai herdar informações da classe pai

```
export class Dog extends Animal {  
  
    constructor(name: string, weight: number){  
        super(name, weight);  
    }  
  
    bark(): void {  
        console.log("au au 🐕");  
    }  
  
}
```

Herança

- `super` é usado para convocar o construtor do pai. Obrigatório quando a superclasse tiver construtor!

```
export class Dog extends Animal {  
  
    constructor(name: string, weight: number){  
        super(name, weight);  
    }  
  
    bark(): void {  
        console.log("au au 🐕");  
    }  
  
}
```

Herança

- Permite economia e reutilização de código, já que uma classe herda atributos e métodos de outra classe
- Isso significa que você pode criar uma classe base com funcionalidades comuns e, em seguida, estender essa classe em classes derivadas, evitando a necessidade de reescrever o mesmo código várias vezes.

```
export class Veiculo {  
    ligar() {  
        console.log("Veículo ligado.");  
    }  
  
    desligar() {  
        console.log("Veículo desligado.");  
    }  
}
```

```
import { Veiculo } from "../Veiculo";  
  
export class Carro extends Veiculo {  
    /* Carro herda ligar() e  
    desligar() da classe Veiculo*/  
    acelerar() {  
        console.log("Carro acelerando.");  
    }  
}
```

```
import { Veiculo } from "../Veiculo";  
  
export class Moto extends Veiculo {  
    /* Moto herda ligar() e  
    desligar() da classe Veiculo*/  
    empinar() {  
        console.log("Moto empinando.");  
    }  
}
```

Herança

- O polimorfismo é uma característica associada à herança que permite que objetos de classes derivadas sejam tratados como **objetos da classe base**. Podemos então usar uma classe base para manipular objetos de várias subclasses.

```
//index.ts
import { Veiculo } from './Veiculo';
import { Carro } from './Carro';
import { Moto } from './Moto';

/*A função realizarAcao aceita um parâmetro do tipo Veiculo,
que é a classe base. No entanto, você pode passar objetos de
classes derivadas (Carro e Moto) como argumentos para essa função.
Isso é possível porque objetos das classes derivadas são tratados
como objetos da classe base, graças ao polimorfismo.*/
function realizarAcao(veiculo: Veiculo) {
    veiculo.ligar();
    veiculo.desligar();
}

const meuCarro = new Carro();
const minhaMoto = new Moto();

realizarAcao(meuCarro); // Trata um objeto Carro como um Veiculo
realizarAcao(minhaMoto); // Trata um objeto Moto como um Veiculo
```


Herança

- A herança permite criar abstrações que modelam relacionamentos hierárquicos entre conceitos. Essa abordagem permite que classes **compartilhem funcionalidades comuns** em uma classe base, ao mesmo tempo em que **possibilita a extensão e especialização** através de classes derivadas.

```
//index.ts
import { Veiculo } from './Veiculo';
import { Carro } from './Carro';
import { Moto } from './Moto';

/*A função realizarAcao aceita um parâmetro do tipo Veiculo,
que é a classe base. No entanto, você pode passar objetos de
classes derivadas (Carro e Moto) como argumentos para essa função.
Isso é possível porque objetos das classes derivadas são tratados
como objetos da classe base, graças ao polimorfismo.*/
function realizarAcao(veiculo: Veiculo) {
    veiculo.ligar();
    veiculo.desligar();
}

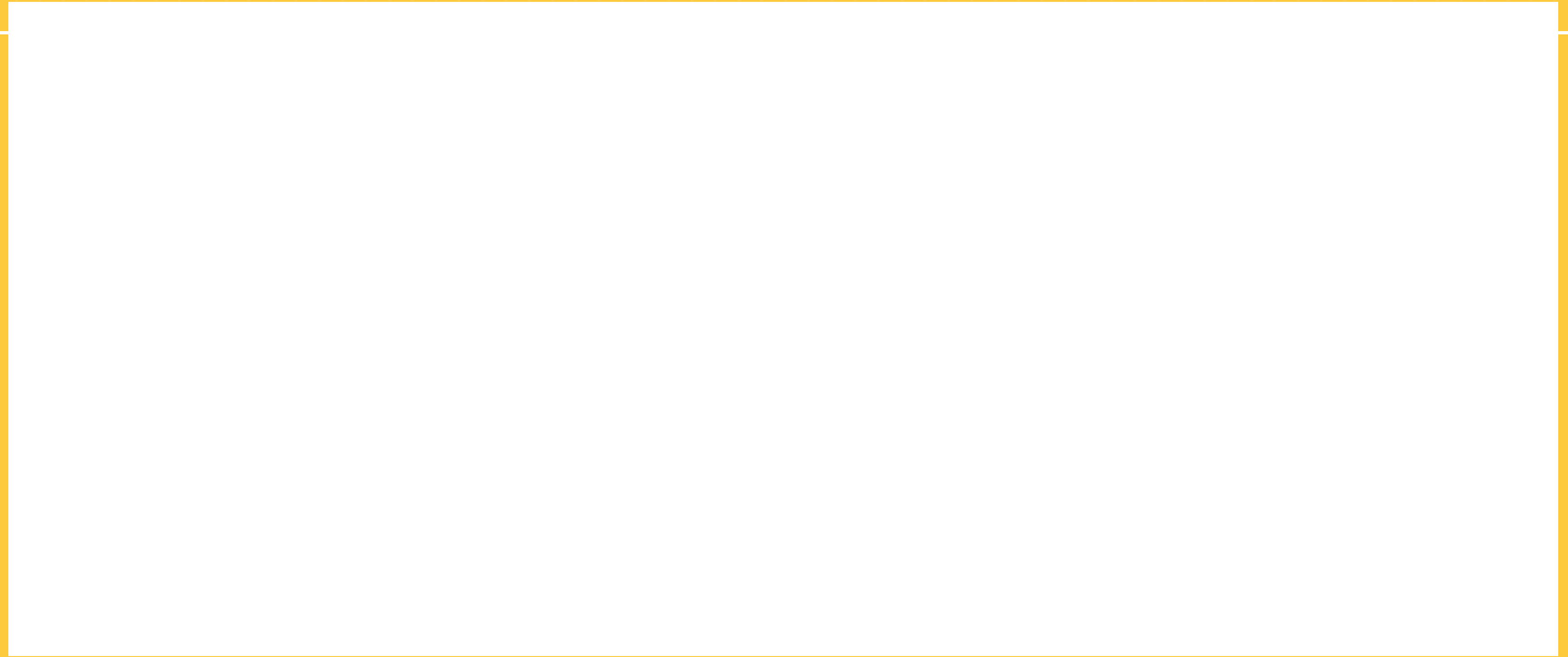
const meuCarro = new Carro();
const minhaMoto = new Moto();

realizarAcao(meuCarro); // Trata um objeto Carro como um Veiculo
realizarAcao(minhaMoto); // Trata um objeto Moto como um Veiculo
meuCarro.acelerar(); // Método específico de Carro
minhaMoto.empinar(); // Método específico de Moto
```

Herança

- Você pode estender as capacidades de uma classe base adicionando novos métodos ou sobrescrevendo métodos existentes em classes derivadas, permitindo a criação de funcionalidades específicas sem afetar a classe base.

Exercício

A large white rectangular area for writing, framed by a thick yellow border. The area is empty, intended for the user to provide their answer to the exercise.

E o encapsulamento?

- Vocês aprenderam dois encapsuladores: public e private
- Existe mais um que só faz sentido no contexto de herança: **protected**
 - Ele cria um método ou uma propriedade que só pode ser acessada pela própria classe e suas filhas

E o encapsulamento?

A palavra-chave `protected` em TypeScript é um modificador de acesso que pode ser aplicado a membros de uma classe (métodos e propriedades). Quando um membro é marcado como `protected`, ele fica acessível dentro da própria classe e também em subclasses (herdeiras) da classe que o contém.

```
✓ export class Person {  
    protected children: number;  
  
    ✓ constructor(children: number) {  
        |     this.children = children;  
        |  
        }  
  
    ✓ protected howManyChildren(): void {  
        |     console.log(`This person has ${this.children} children.`);  
        |  
        }  
    }
```



```
import { Person } from "../Person";  
class Father extends Person {  
  falar(): void {  
    console.log("This is the father.");  
    this.howManyChildren; // Pode acessar membros protected da classe pai  
  }  
}  
  
const pai = new Person(4);  
pai.howManyChildren(); // Não pode acessar fora da classe
```

E o encapsulamento?

A tabela abaixo resume as características dos três encapsuladores que vimos

	Pode acessar dentro da classe?	Filhas podem acessar?	Pode acessar fora da classe?
private	✓	✗	✗
protected	✓	✓	✗
public	✓	✓	✓

Herança

- Recrie o exercício anterior, agora utilizando protected