Programação Orientada a Objetos

Classes e Encapsulamento

Labenu_

Introdução

O que é a Programação Orientada a Objetos (POO ou OOP)

- Paradigma de programação
- Surgiu na década de 1970 e utiliza objetos como base para solução dos problemas
    - **objeto**
        - representa algo do mundo real
        - exemplo no contexto de uma loja:
            - cada produto será um objeto

Introdução

- Paradigmas **não são linguagens**
- Linguagens podem implementar **múltiplos paradigmas**
- Utilizar um paradigma **não significa** abandonar as ferramentas de outros

Introdução

- Os paradigmas nos trazem formas diferentes para solucionar o mesmo problema
- O JS **não implementa** o paradigma clássico de POO
  - é possível criar objetos sem pré-definir uma classe
  - utiliza delegação com Prototypes para aplicar herança
- A POO possui **4 pilares (princípios)**:
  - Encapsulamento, Herança, Polimorfismo e Abstração

Objetos

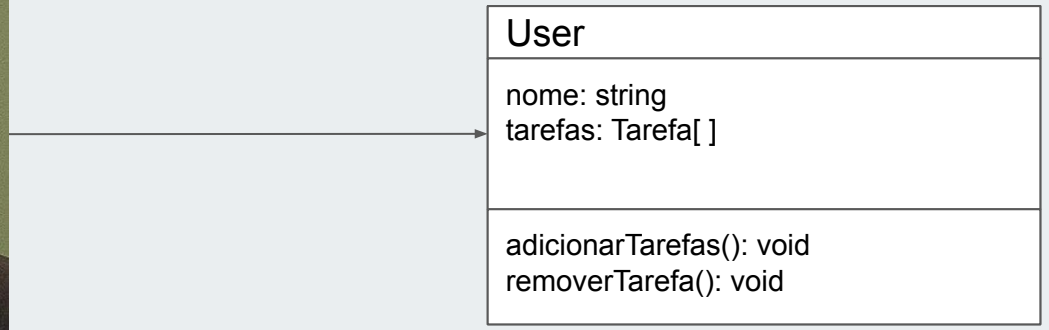
- Criados a partir de uma **classe**
- Representam algo do mundo real
- São independentes entre si
- Suas propriedades são chamadas de atributos
- Suas funções são chamadas de métodos

Classes

- Modelam um template de algo do mundo real
- Instanciam / criam / inicializam objetos através do construtor*
- Cada classe pode gerar múltiplos objetos
- Podem ser utilizadas para tipagem no Typescript

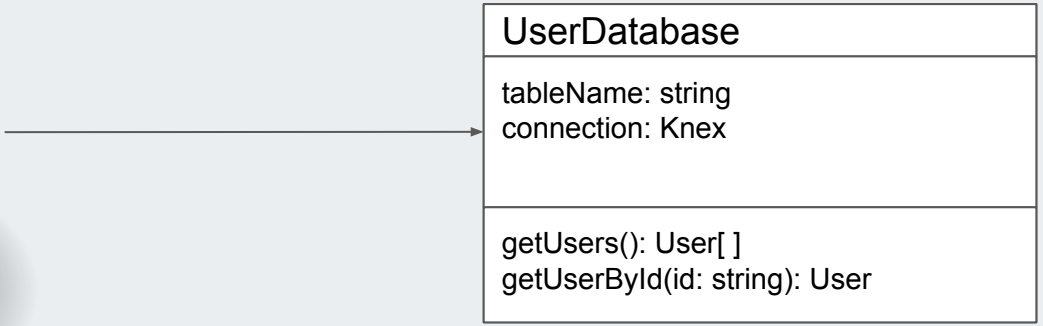
**existe um tipo de classe que não pode ser instanciada*

Exemplo 🎲


User
nome: string
tarefas: Tarefa[ ]
adicionarTarefas(): void
removerTarefa(): void

Exemplo 🎲


MySQL


UserDatabase
tableName: string
connection: Knex
getUsers(): User[ ]
getUserById(id: string): User

Criando uma classe

Labenu_

Sintaxe na forma mais verbosa

```
export class User {

}
```

Sintaxe na forma mais verbosa

```
export class User {
    // atributos
    public id: string;
    public email: string;
    public password: string;

}
```

Sintaxe na forma mais verbosa

```
export class User {
    // atributos
    public id: string;
    public email: string;
    public password: string;

    // método construtor
    constructor(
        id: string,
        email: string,
        password: string
    ) {
        this.id = id
        this.email = email
        this.password = password
    }


}
```

Sintaxe na forma mais verbosa

```
export class User {
    // atributos
    public id: string;
    public email: string;
    public password: string;

    // método construtor
    constructor(
        id: string,
        email: string,
        password: string
    ) {
        this.id = id
        this.email = email
        this.password = password
    }

    // outros métodos (falaremos mais na parte de encapsulamento)
}
```

Sintaxe na forma abreviada

```
export class User {
    // atributos foram abreviados nos parâmetros do construtor
    // eles ainda existem e estão com os mesmos nomes

    // método construtor
    constructor(
        public id: string,
        public email: string,
        public password: string
    ) {
        this.id = id
        this.email = email
        this.password = password
    }

    // outros métodos (falaremos mais na parte de encapsulamento)
}
```

This 🎯

- Sim, ele voltou!
- A keyword (palavra-chave) **this** é usada para **referenciar** os **atributos** e **métodos** daquela **instância**
- Será utilizada no construtor e nos métodos para acessar e manipular os dados do objeto

Instanciando um objeto

Labenu_

Instanciando (Construindo) 🚧

- A criação de novas instâncias é feita com a *keyword* **new**

```
const user = new User(
    Date.now().toString(),
    "astrodev@gmail.com",
    "abc123"
)

console.log(user)
```

Instanciando (Construindo) 🚧

- Comparando com o **constructor** da classe

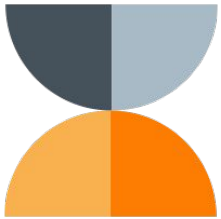
```
const user = new User(
    Date.now().toString(),
    "astrodev@gmail.com",
    "abc123"
)

console.log(user.id)
```

```
constructor(
        public id: string,
        public email: string,
        public password: string
    ) {
        this.id = id
        this.email = email
        this.password = password
    }
```

Pausa para relaxar 😴

10 min



- **Objetos** possuem **atributos** e **métodos**
- **Classes** determinam formato dos **objetos** e podem ser usadas como **tipos** de variáveis
- **Construtor** serve para inicializar objetos com atributos definidos
- Para criar uma nova instância da classe, usa-se a *keyword* **new**

Exercício 1

Transforme o *type* **Product** do **Labecommerce** em uma *classe*.

Refatore também o código onde o type era utilizado, agora para uma instância.

Encapsulamento

Labenu_

Conceito 

- Encapsular na prática significa proteger o acesso aos dados do objeto
- Da forma como fizemos, deixando tudo público, qualquer dev pode acessar o objeto e modificar seus dados
- Os atributos da classe serão privados (**private**) e alguns métodos serão públicos (**public**)

Public vs private 📢❌🔒

- Encapsular é utilizar as *keywords* **public** e **private**
- Por padrão toda variável é **public**, o que significa que pode ser acessada fora do escopo da classe
- Variáveis privadas só podem ser acessadas de dentro da própria classe, normalmente utilizando a keyword **this**
- Existe outro tipo de encapsulamento que utiliza o princípio da herança! Veremos ele amanhã

Exemplo

```
export class User {
    constructor(
        private id: string,
        private email: string,
        private password: string
    ) {
        this.id = id
        this.email = email
        this.password = password
    }
}
```

```
const user = new User(
    Date.now().toString(),
    "astrodev@gmail.com",
    "abc123"
)

console.log(user.id)
// Property 'id' is private and only accessible within class 'User'.
```

Getters e Setters

- É recomendado tornar **todos os atributos privados** e controlar o acesso por **métodos públicos**
- Esses métodos são chamados de **getters** (para pegar o atributo) e **setters** (para definir)
- Isso garante consistência e extensibilidade

Exemplo com getter da id

```
class User {
    constructor(
        private id: string,
        private email: string,
        private password: string
    ) {
        this.id = id
        this.email = email
        this.password = password
    }

    public getId() {
        return this.id
    }
}
```

```
const user = new User(
    Date.now().toString(),
    "astrodev@gmail.com",
    "abc123"
)

console.log(user.getId())
// "1658689651198"
```

Exemplo com setter da id

```
class User {
    constructor(
        private id: string,
        private email: string,
        private password: string
    ) {
        this.id = id
        this.email = email
        this.password = password
    }

    public setId(newId: string) {
        this.id = newId
    }
}
```

```
const user = new User(
    Date.now().toString(),
    "astrodev@gmail.com",
    "abc123"
)

user.setId("bananinha")
console.log(user.getId())
// "bananinha"
```

Exercício 2

Torne as propriedades do Product **privadas**.
Adicione getters e setters necessários.

É isso por hoje!

Labenu_

Obrigado!