Programação Orientada a Objetos
Herança, static e abstract

Labenu_

Herança

Labenu_

Herança 💰

- **Reutilização** de lógica e código (atributos e métodos)
- A classe **filha** herda da classe **pai**
- A **subclasse** herda da **superclasse**
- A classe **derivada** herda da classe **base**
- As classes podem ser **filhas** e **parentes** ao **mesmo tempo**

Herança 💰

- Para exemplificar, observe as estruturas de Estudante e Docente

Estudante
id: string nome: string email: string dataNasc: Date turma: string hobbies: string[]

Docente
id: string nome: string email: string dataNasc: Date turma: string especialidades: string[]

Herança 💰

- Vamos isolar tudo que é comum para montar uma estrutura genérica e criar a classe **Pessoa**

Pessoa
id: string nome: string email: string dataNasc: Date turma: string

Herança 💰 - Forma abreviada antiga

```
export class Pessoa {
    constructor(
        private id: string,
        private nome: string,
        private email: string,
        private dataNasc: Date,
        private turma: string
    ) {
        this.id = id
        this.nome = nome
        this.email = email
        this.dataNasc = dataNasc
        this.turma = turma
    }

    // métodos getters e setters...
}
```

Herança 💰 - Forma abreviada final

```
export class Pessoa {
    constructor(
        private id: string,
        private nome: string,
        private email: string,
        private dataNasc: Date,
        private turma: string
    ) {
        // não precisa atribuir manualmente!
    }

    // métodos getters e setters...
}
```

Herança 💰

Declarando *Estudante*

extends
indica que a classe **Estudante** vai herdar informações da classe **Pessoa**

super
chama o construtor do pai. Obrigatório quando a superclasse tiver construtor

```
export class Estudante extends Pessoa {
    constructor(
        id: string,
        nome: string,
        email: string,
        dataNasc: Date,
        turma: string,
        private hobbies: string[]
    ) {
        super(
            id,
            nome,
            email,
            dataNasc,
            turma
        )

        this.hobbies = hobbies // pode ser abreviado
    }

    // método getter e setter de hobbies
}
```

Herança 💰

Declarando *Docente*

extends
indica que a classe **Docente** vai herdar informações da classe **Pessoa**

super
chama o construtor do pai. Obrigatório quando a superclasse tiver construtor

```
export class Docente extends Pessoa {
    constructor(
        id: string,
        nome: string,
        email: string,
        dataNasc: Date,
        turma: string,
        private especialidades: string[]
    ) {
        super(
            id,
            nome,
            email,
            dataNasc,
            turma
        )

        this.especialidades = especialidades // pode ser abreviado
    }

    // método getter e setter de especialidades
}
```

Herança 💰

Inicializando *Estudante*

```
const estudante = new Estudante(
    "101",
    "Fulana",
    "fulana@gmail.com",
    new Date("1999/12/15"),
    "Aragon",
    ["assistir filmes", "academia", "codar"]
)

console.log(estudante.getEmail())
```

Herança 💰

Inicializando *Docente*

```
const docente = new Docente (
    "201",
    "Astrodev",
    "astrodev@gmail.com",
    new Date("1980/01/01"),
    "Aragon",
    ["POO", "TS", "JS"]
)

console.log(docente.getEmail())
```

Herança + Encapsulamento

Labenu_

Herança + Encapsulamento 🏦

- Ontem vimos dois encapsuladores: **public** e **private**
- Existe mais um que só faz sentido no contexto de herança: o encapsulador **protected**
    - delimita o atributo ou método para ser acessível somente pela **própria classe** e **suas filhas (e descendentes)**

Exemplo - Refatoração da connection

```
export class BaseDatabase {
   protected connection = knex({
      client: "mysql",
      connection: {
         host: process.env.DB_HOST,
         port: 3306,
         user: process.env.DB_USER,
         password: process.env.DB_PASSWORD,
         database: process.env.DB_DATABASE,
         multipleStatements: true
      },
   });
}
```

protected
indica que o atributo connection só pode ser acessado pela classe e seus **descendentes** (herança)

Exemplo - Refatoração da connection

```
export class UserDatabase extends BaseDatabase {
   public async getAllUsers() {
      console.log(this.connection)
   }
}
```

extends

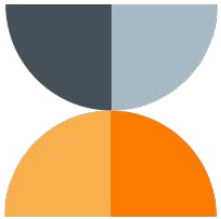
indica que a classe **UserDatabase é filha** da BaseDatabase

a UserDatabase tem **acesso aos dados** da BaseDatabase

Herança + 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** é uma estrutura que permite que aproveitemos classes que já criamos para novas

- Para criar um filho/subclasse, usamos a keyword **extends**

- Se quisermos criar um construtor para nossa classe, nós precisamos convocar o construtor da classe pai, usando **super()**

- O encapsulador **protected** permite que declaremos métodos e propriedades que possam ser acessados pela **própria** classe e suas **filhas**

Variáveis estáticas

Labenu_

Contexto

- Objetos são independentes
  - atributos e métodos de cada instância estão separados de outras instâncias
  - não existe compartilhamento de dados entre instâncias
  - **muito uso de recursos do sistema (memória)**

Contexto

- Surge então o conceito de **variáveis estáticas**
  - dados compartilhados entre as instâncias (**constantes**)
  - não necessita instanciar para utilizá-las
  - recurso compartilhado (**economia de memória**)

JSON.parse()
JSON.stringify()

Date.now()
Math.random()

Exemplo - Refatoração da BaseDatabase

```
export class BaseDatabase {
   protected static connection = knex({
      client: "mysql",
      connection: {
        host: process.env.DB_HOST,
        port: 3306,
        user: process.env.DB_USER,
        password: process.env.DB_PASSWORD,
        database: process.env.DB_DATABASE,
        multipleStatements: true
      },
    });
}
```

static

indica que o atributo connection é **compartilhado** entre as instâncias

a connection só pode ser acessada via classe (**sem instanciar**)

Exemplo - Refatoração da UserDatabase

```
export class UserDatabase extends BaseDatabase {
   public static TABLE_USERS = "Labe_Users"

   public async getAllUsers() {
      const result = await BaseDatabase
         .connection(UserDatabase.TABLE_USERS)
         .select()

      return result
   }
}
```

UserDatabase.TABLE_USERS

essa é a forma que acessamos uma variável estática

nome da classe e atributo/método

BaseDatabase.connection

essa é a forma que acessamos uma variável estática

nome da classe e atributo/método

Classes abstratas

Labenu_

Contexto

- Voltando ao exemplo de Estudantes e Docentes
    - criamos uma classe genérica Pessoa
    - faz sentido **instanciar essa classe?**
- Quando **não for necessário** instanciar uma classe, é recomendado defini-la como **abstrata**
- Classes abstratas são utilizadas em **heranças**

Exemplo - Refatoração da Pessoa

```
export abstract class Pessoa {
    constructor(
        private id: string,
        private nome: string,
        private email: string,
        private dataNasc: Date,
        private turma: string
    ) { }

    // métodos getters e setters...
}
```

abstract
indica que a classe **Pessoa**
é do tipo abstrata
não pode ser instanciada

ela será herdada

Exemplo - Refatoração da BaseDatabase

```
export abstract class BaseDatabase {
   protected static connection = knex({
      client: "mysql",
      connection: {
         host: process.env.DB_HOST,
         port: 3306,
         user: process.env.DB_USER,
         password: process.env.DB_PASSWORD,
         database: process.env.DB_DATABASE,
         multipleStatements: true
      },
   });
}
```

abstract
indica que a classe **BaseDatabase** é do tipo abstrata
não pode ser instanciada

ela será herdada

É isso por hoje!

Labenu_

Obrigado!