Angular

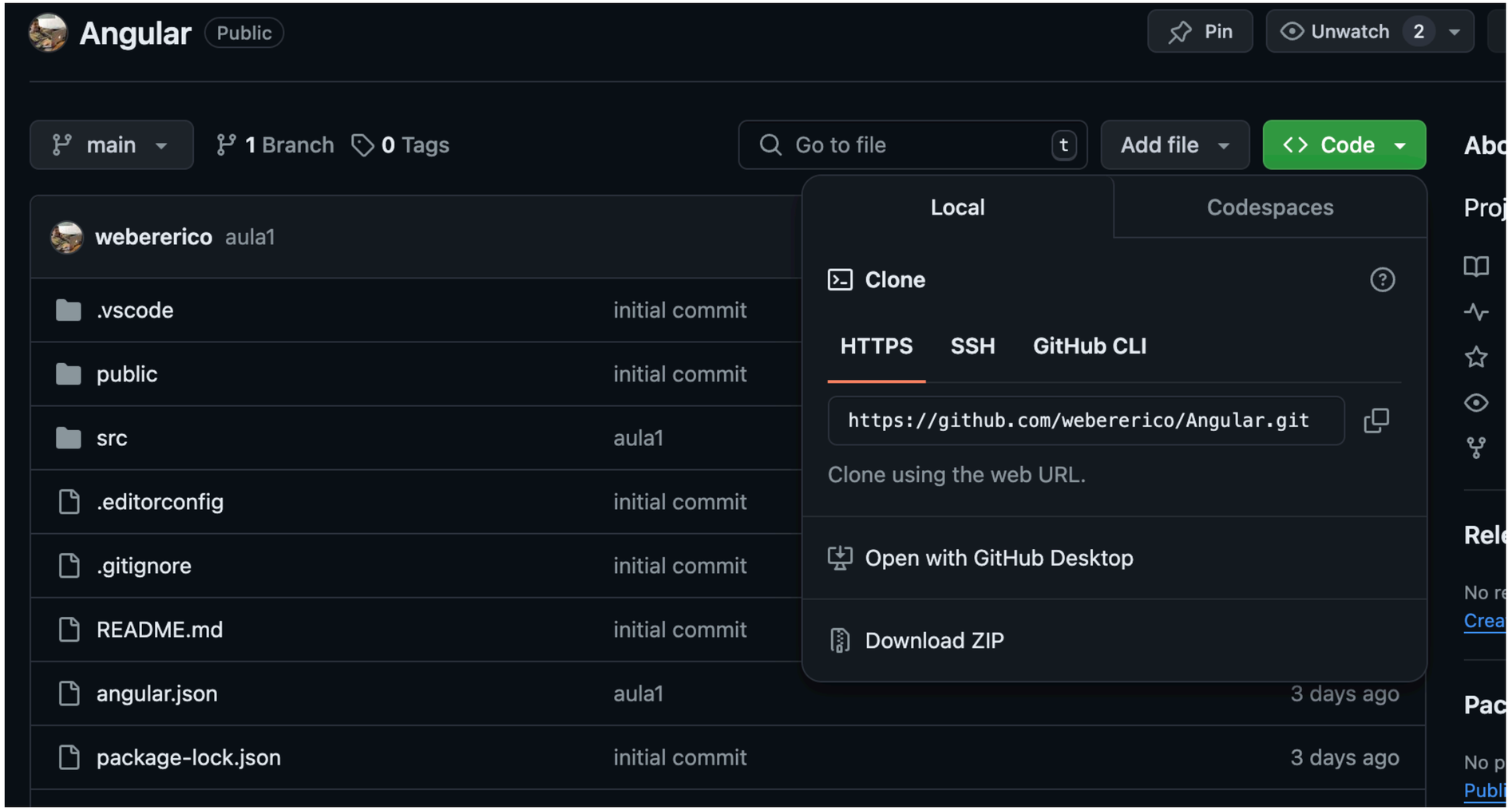
Érico Rosiski Weber - 22/07/2024

Aula 2 - Conceitos básicos

- Interporlação e estilos
- Compartilhamento de componentes e Diretivas
- Renderização condicional
- Eventos e Event Emitter
- Interfaces e PipeOperators
- Two Way data binding
- Services conceito

Git clone

https://github.com/webererico/Angular.git



Documentation

- https://v17.angular.io/docs

- https://angular.dev/overview

Interpolação de dados

Dentro do componente:

- Propriedades da classe no arquivo .ts são nossas variáveis acessíveis no template - arquivo .html
- Para utilizar as variáveis no arquivo .html utilizamos a escrita: {{dado}}

No arquivo .ts

```ts
export class HomeComponent {
  title: String = "Home";
  age: number = 10;
}
```

No arquivo .html

```html
<p>Title: {{title}}</p>
<p> Age : {{age}}</p>
```

Estilos no Angular

Duas maneiras:

- **Global:**
  - style.css no diretório source: fontes, padrões visuais utilizados em todo projeto

```
* {
    margin:0;
    padding: 20;
    font-family: Helvetica;
}
```

- **Scoped:**
  - Nível do componente, arquivo .css dentro do componente.
    (Criado com o comando generate)

```
a {
    color: green;
}

.title {
    background-color: blue;
    padding: 15;
    color: white;
}
```

Compartilhar dados entre componentes

- Dados compartilhados entre componente pai e componente filho
  - Decorator @input: entregar o dado para o template

Componente pai

```
You, 3 minutes ago | 2 authors (Érico Rosiski Weber and one other)
@Component({
  selector: 'app-home',
  standalone: true,
  imports: [HomeBodyComponent],
  templateUrl: './home.component.html',
  styleUrl: './home.component.css'
})
```

Componente filho

```
@Component({
  selector: 'app-home-body',
  standalone: true,
  imports: [],
  templateUrl: './home-body.component.html',
  styleUrl: './home-body.component.css'
})
export class HomeBodyComponent {
  welcomeMessage: String = 'Bem-vindo a locadora'
}
```

Diretivas

Directives no Angular 18

Directives no Angular 17

- Aplicar estilos a um elemento
- Representa uma ação de estilo no template.
- 3 tipos:
    - Componente -> usada com o template
    - Atributo -> Altere a aparência ou o comportamento de um elemento, componente ou outra diretiva.
        - NgClass, NgStyle, NgModel
    - Estruturais -> As diretivas estruturais são responsáveis pelo layout HTML. Eles moldam ou remodelam a estrutura, normalmente adicionando, removendo e manipulando os elementos hospedeiros aos quais estão anexados.
        - Ngif, NgFor, NgSwitch

src/app/app.component.ts

```
  currentStyles: Record<string, string> = {};
...
  setCurrentStyles() {
    // CSS styles: set per current state of component properties
    this.currentStyles = {
      'font-style': this.canSave ? 'italic' : 'normal',
      'font-weight': !this.isUnchanged ? 'bold' : 'normal',
      'font-size': this.isSpecial ? '24px' : '12px',
    };
  }
```

To set the element's styles, add an `ngStyle` property binding to `currentStyles`.

src/app/app.component.html

```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large (24px).
</div>
```

Exemplo uso diretiva NgStyle

Renderização condicional

- Também é diretiva, mas com foco em condição
- NgIf, NgSwitch

```
<div class="class">
    <h1>If Render</h1>
    <div *ngIf="canShow">Estamos mostrando isso de acordo com a condicao</div>
    <h2 *ngIf="name==='Erico'">Olá {{name}}</h2>
</div>
```

Exemplo de uso do ngIf

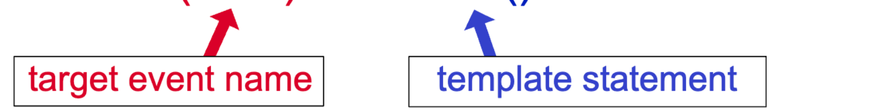
ng-template é uma diretiva Angular que serve como espaço reservado para renderização de conteúdo HTML. Ao contrário de outros elementos HTML, ele não exibe nada por si só. Em vez disso, serve como um modelo para gerar conteúdo dinamicamente, tornando-o uma ferramenta versátil para vários cenários.

Eventos

= disparar algum método através da interface no controller (arquivo html)

- Ex: clique em um botão
- Sintaxe:
  (Click) = “algumaFuncao()”

```
<button (click)="onSave()">Save</button>
```



Normalmente, interações com o usuário no front são interações realizadas através de eventos.

Um bom exemplo é listar dados de um banco de dado através de um botão de buscar

Ao clicar em determinada tecla, disparar um evento:

```
<input (keydown.shift.t)="onKeydown($event)" />
```

Eventos entre componentes (Event Emitter)

= eventos entre componente pai e filho

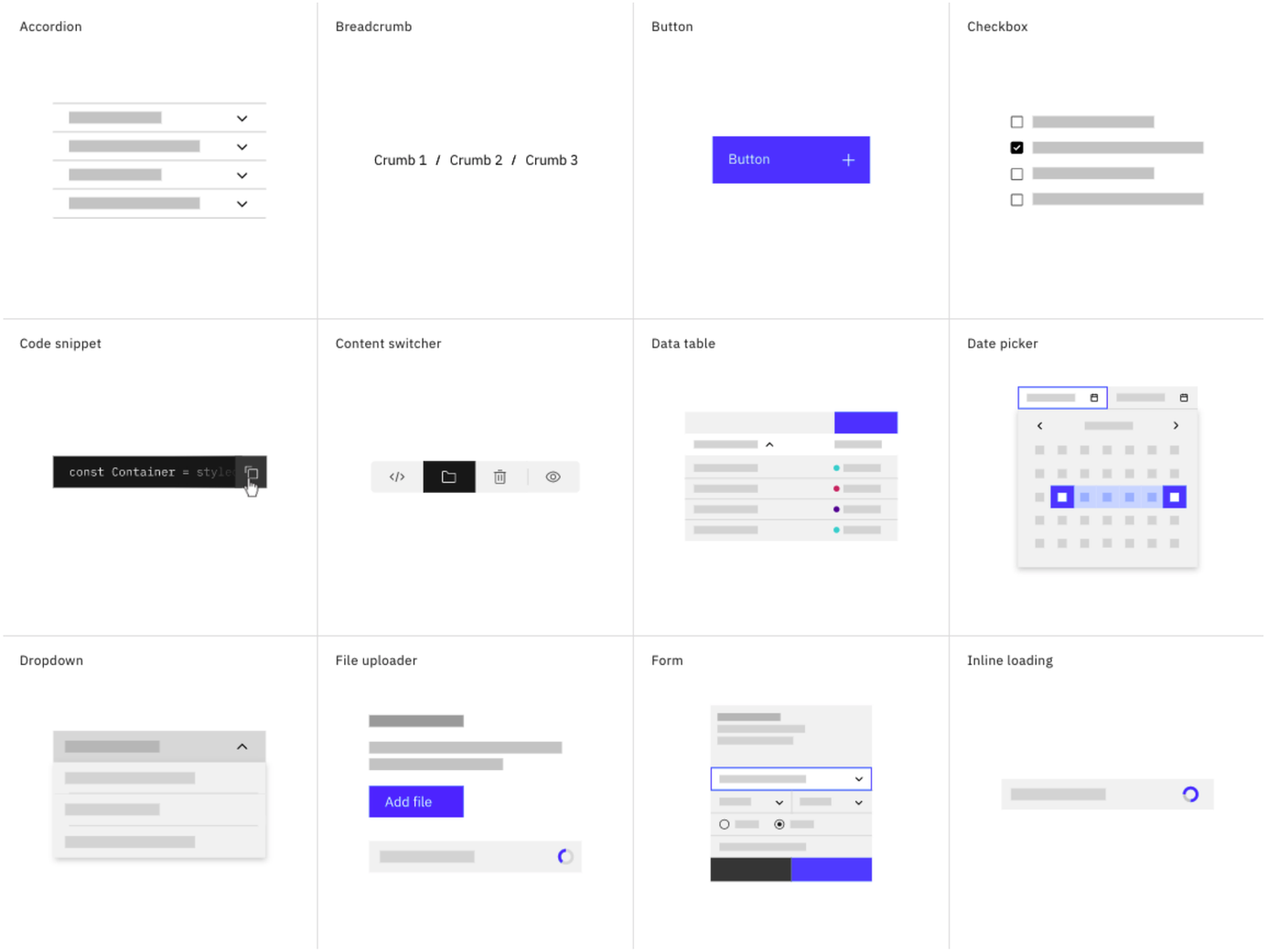
- Ex: Home possui componentes reutilizáveis (buttonComponent), mas a lógica está na HomeComponent
  - Emitir um evento no componente filho que dispara uma lógica no componente pai
- Sintaxe:
  (emit) = “onEmit()”

```
@Component({
  selector: 'zippy',
  template: `
  <div class="zippy">
    <div (click)="toggle()">Toggle</div>
    <div [hidden]="!visible">
      <ng-content></ng-content>
    </div>
 </div>`})
export class Zippy {
  visible: boolean = true;
  @Output() open: EventEmitter<any> = new EventEmitter();
  @Output() close: EventEmitter<any> = new EventEmitter();

  toggle() {
    this.visible = !this.visible;
    if (this.visible) {
      this.open.emit(null);
    } else {
      this.close.emit(null);
    }
  }
}
```

Design System

- Event emitter ajuda a generalizar funções de componentes



Renderização de listas

Loops

- NgFor = Também uma diretiva

```
<div class="class">
    <h1> Lista Carros</h1>
    <ul>
        <li *ngFor="let car of cars">
            Nome: {{car.name}} Marca: {{car.brand}}
        </li>
    </ul>
</div>
```

```
export class ListRenderComponent {
  cars = [
    {
      name: "208", brand: "Peugeot"
    },
    {
      name: "Pulse", brand: "Fiat"
    },
    {
      name: "Kwid", brand: "Reault"
    },
    {
      name: "Renegade", brand: "Jeep"
    }
  ]
}
```

Interfaces

Recurso do TypeScript mas utilizado no Angular

- Interfaces
    - Transformar dados em classes que representam a entidade.
    - Impor contratos estruturados

```
export interface Car{
    name: string,
    brand: string,
}
```

Pipe Operators

Formatar Strings nos templates

https://angular.dev/guide/pipes

- Sintaxe:
  {{dado | pipeOperator }}

- Poupar regras do CSS e manipulação de Strings *transformações para internacionalização (i18n)*

  - `DatePipe` : Formata um valor de data de acordo com as regras de localidade.
  - `UpperCasePipe` : Transforma o texto todo em letras maiúsculas.
  - `LowerCasePipe` : Transforma o texto todo em letras minúsculas.
  - `CurrencyPipe` : Transforma um número em uma sequência de moeda, formatada de acordo com as regras locais.
  - `DecimalPipe` : Transforma um número em uma string com um ponto decimal, formatada de acordo com as regras locais.
  - `PercentPipe` : Transforma um número em uma sequência de porcentagem, formatada de acordo com as regras locais.
  - `AsyncPipe` : Assinar e cancelar a assinatura de uma fonte assíncrona, como um observável.
  - `JsonPipe` : Exibe uma propriedade de objeto componente na tela como JSON para depuração.

Two way data binding

Mais usado para formulários

- Alterar propriedades e template com o preenchimento de inputs
- Utiliza o FormsModule (necessário importar)
- “ngModel"

Services

Compartilhamento de Dados e Lógica & Injeção de Dependência:

Serviços são usados para centralizar e compartilhar dados, lógica de negócios ou funções entre componentes diferentes da aplicação Angular.

Angular usa um sistema de injeção de dependência para injetar serviços em componentes, diretivas, pipes ou outros serviços. Isso facilita a reutilização e a manutenção do código, além de melhorar a testabilidade.

Implementação de Funcionalidades Comuns e Separação de Preocupações:

Serviços são ideais para encapsular funcionalidades comuns, como acesso a APIs HTTP, manipulação de dados, autenticação de usuários, etc.

Utilizando serviços, é possível separar as preocupações da apresentação (componentes) das preocupações de negócios ou dados. Isso promove um código mais limpo e organizado.

Padrão Singleton:

Por padrão, os serviços em Angular são singletons, ou seja, uma única instância do serviço é criada e compartilhada por toda a aplicação. Isso garante que os dados e estados mantidos pelo serviço sejam consistentes em toda a aplicação.

Models (OOP)

Programação Orientada a Objetos

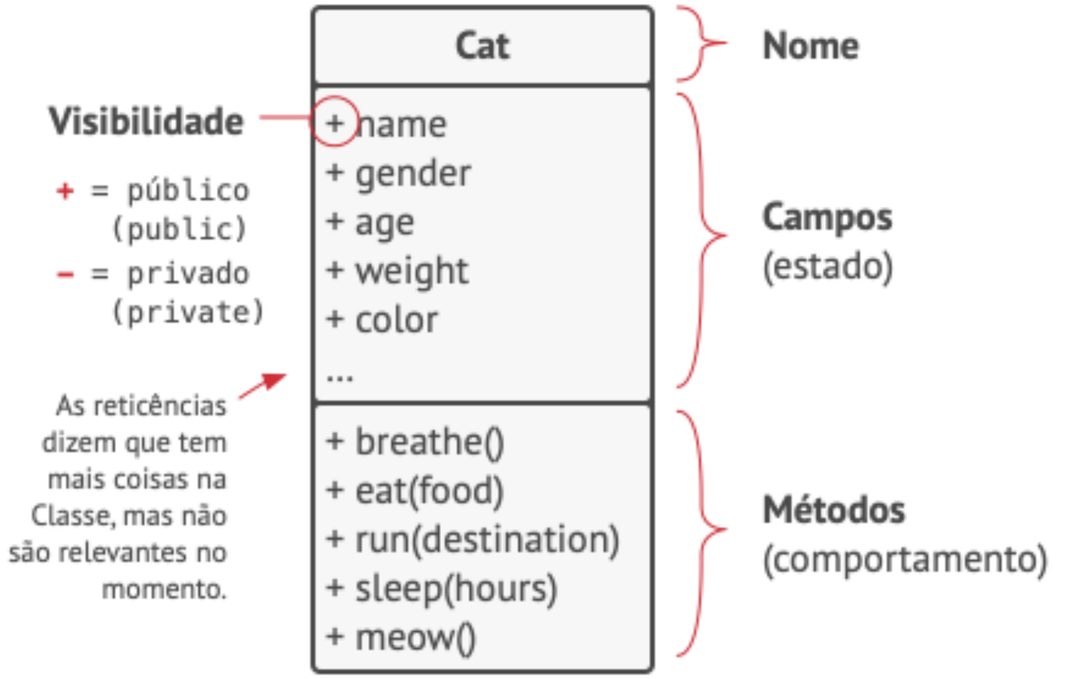
OOP

Orientação a objetos é um paradigma aplicado na programação que consiste na interação entre diversas unidades chamadas de objetos.

Usamos a orientação a objetos para nos basear na vida real e resolver problemas de software, ou pelo menos tentamos. Ela acaba sendo uma base inclusive para outros paradigmas.

Não é ligado a linguagem ou lógica e sim a um paradigma de pensamento

4 pilares fundamentais
herança, encapsulamento, abstração e polimorfismo.




```
class CatModel{
  name: String = 'Alfredo';
  gender: String = 'Male';
  age: number = 3;
  weight: number = 2.7;
  color: string = 'black';

  breath () : void{

  }

  eat () {

  }
  // ...
}
```

OOP - Encapsulamento

Aplicando o conceito de encapsulamento, fazemos o agrupamento das coisas que fazem sentido estarem juntas, para podermos organizar e reutilizar melhor nosso código.

```
int idade = 34;

...

public void VerificaIdade() {
    if(idade > 18) ...
}

public void AtribuiIdade(int idade) {
    idade = idade;
}
```

As variáveis passam então a ser o que chamamos de **propriedades** e as funções passam a ser o que chamamos de **métodos**. Esta composição dá origem ao nosso objeto e aplica o conceito de encapsulamento.

```
public class Aluno {
    public int Idade { get; set; }

    public Aluno(int idade) {
        Idade = idade;
    }

    public void VerificaIdade() {
        if(idade > 18) ...
    }
}
```

OOP - Herança

O conceito de **herdar** é literalmente ser uma de outra classe com algumas características adicionais.

```
public class Pagamento {
    public DateTime Vencimento { get; set; }
}
```

```
public class PagamentoBoleto : Pagamento {
    public string CodigoBarras { get; set; }
}
```

Neste caso, temos uma nova classe, chamada **PagamentoBoleto** que herda as características da sua classe pai **Pagamento**. Desta forma, na classe **PagamentoBoleto** temos tanto a data de vencimento quanto o código de barras

OOP - Abstração

Tanto a propriedade quanto o método são usados exclusivamente no boleto, e caso precisem ser alterados, se consumidos externamente, causariam uma refatoração em vários outros componentes.

```
public class PagamentoBoleto : Pagamento {
    public int digitoVerificador = 0;

    public string CodigoBarras { get; set; }

    public int CalcularDigitoVerificador() {
        ...
    }
}
```

Desta forma, escondemos tudo que não é necessário o mundo externo ao nosso objeto saber, assim ficamos mais confortáveis com as mudanças, pois elas afetam somente o nosso objeto.

```
public class PagamentoBoleto : Pagamento {
    private int digitoVerificador = 0;

    public string CodigoBarras { get; set; }

    private int CalcularDigitoVerificador() {
        ...
    }
}
```

OOP - Polimorfismo

Novamente no caso dos pagamentos, embora cada pagamento seja pertinente a uma operadora distinta, ainda temos comportamentos que serão padrão em todos eles.

Desta forma, escondemos tudo que não é necessário o mundo externo ao nosso objeto saber, assim ficamos mais confortáveis com as mudanças, pois elas afetam somente o nosso objeto.

```
public class Pagamento {
    public bool PodeSerPago() {
        ...
    }
}
```

```
public class Pagamento {
    public virtual bool PodeSerPago() {
        ...
    }
}

public class PagamentoBoleto : Pagamento {
    public override bool PodeSerPago() {
        ...
    }
}
```

Exemplo

No contexto de um sistema de gerenciamento escolar, alguns objetos principais seriam: **Aluno, Professor, Disciplina e Turma.**

- Abstração: Ao modelar a classe **Aluno**, podemos abstrair características essenciais como nome, idade e matrícula, sem nos preocuparmos com detalhes irrelevantes para a funcionalidade principal do sistema, como a cor do cabelo.
- Encapsulamento: A classe **Professor** encapsula atributos como nome e disciplinas lecionadas, além de métodos como "adicionarNota()" e "calcularMédia()", protegendo esses dados e comportamentos de acessos não autorizados.
- Herança: A classe **Disciplina** pode herdar características básicas de uma classe mais genérica, como **DisciplinaBase**, aproveitando métodos como "obterCargaHorária()" e "definirEmenta()", mas também pode ter métodos específicos como "calcularMédiaTurma()" para calcular a média de uma turma inteira.
- Polimorfismo: O método "calcularMédia()" pode ser implementado de maneira diferente na classe **Aluno** e na classe **Turma**, refletindo a necessidade de calcular a média individual de um aluno versus a média geral de uma turma.

Esses pilares da programação orientada a objetos permitem criar um sistema de gerenciamento escolar flexível, modular e de fácil manutenção, refletindo as características reais e operações do ambiente escolar de maneira estruturada e eficiente.

Exercícios

CRIAR UM PROJETO ANGULAR NOVO

Exercício 1: Vinculação de dados

Crie um componente AngularJS que vincule dados de um controlador ao front end. Exibe uma lista de itens com a diretiva ng-repeat.

Exercício 2: Tratamento de Eventos

Implemente um componente AngularJS que lide com as interações do usuário, como clique em botão ou envio de formulário. Use as diretivas ng-click e ng-submit para manipulação de eventos.

Exercício 3: vinculação de dados bidirecional

Crie um formulário com campos de entrada que demonstrem a ligação bidirecional de dados entre a visualização e o modelo. Use a diretiva ng-model para ligação bidirecional.

Exercício 4: Diretivas Condicionais

Desenvolva uma diretiva AngularJS personalizada que encapsule uma parte reutilizável da lógica da UI. Implemente a diretiva em um componente front-end.

Próxima Aula

- Angular Router
- Requisições HTTP (API)
- Integrando um template
- Start de um projeto exemplo
- Integrando

Avalie essa aula:
Formulário avaliação Aula 2