O uso de `@Input` **com getters e setters** é uma prática poderosa no Angular, pois permite executar lógica adicional sempre que o valor de uma propriedade de entrada for alterado. Isso é útil para validações, transformação de dados, ou qualquer outra manipulação necessária antes de armazenar o valor.

Abaixo está uma explicação detalhada com exemplos e possíveis casos de uso.

Exemplo Simples: Transformando Valores com Getters e Setters

```
import { Component, Input } from '@angular/core';

@Component({
  selector: 'app-example',
  template: `<p>Valor transformado: {{ internalValue }}</p>`,
})
export class ExampleComponent {
  private _value: string = '';

  // Getter para acessar o valor interno
  get internalValue(): string {
    return this._value;
  }

  // Setter que é chamado sempre que a propriedade de entrada mudar
  @Input()
  set value(newValue: string) {
    this._value = newValue.toUpperCase(); // Transforma o valor para
maiúsculas
  }
}
```

Uso no Componente Pai

```
<app-example [value]="'texto inicial'"></app-example>
```

Resultado na Tela
Valor transformado: TEXTO INICIAL

Exemplo 2: Validação de Entrada

Um caso prático seria validar se o valor recebido é aceitável.

```
@Component({
  selector: 'app-validated-input',
  template: `
    <p *ngIf="hasError" style="color: red;">Valor inválido!</p>
    <p>Valor válido: {{ validatedValue }}</p>
  `,
})
export class ValidatedInputComponent {
  private _value: number = 0;
  hasError: boolean = false;

  // Getter para o valor validado
  get validatedValue(): number {
    return this._value;
  }

  // Setter para validar o valor
  @Input()
  set value(newValue: number) {
    if (newValue < 0 || newValue > 100) {
      this.hasError = true; // Marca como inválido
    } else {
      this.hasError = false; // Marca como válido
      this._value = newValue; // Armazena o valor
    }
  }
}
```

Uso no Componente Pai

```
<app-validated-input [value]="120"></app-validated-input>
<app-validated-input [value]="50"></app-validated-input>
```

Resultado na Tela
Valor inválido!
Valor válido: 50

Exemplo 3: Disparando Eventos no Setter

Além de transformar ou validar o valor, você também pode disparar eventos ou executar ações adicionais no `set`.

```
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-notifier',
  template: `<p>Valor atualizado: {{ currentValue }}</p>`,
})
export class NotifierComponent {
  private _value: string = '';

  @Output() valueUpdated = new EventEmitter<string>();

  // Getter
  get currentValue(): string {
    return this._value;
  }

  // Setter com emissão de evento
  @Input()
  set value(newValue: string) {
    this._value = newValue;
    this.valueUpdated.emit(`Novo valor: ${newValue}`); // Dispara um evento ao alterar o valor
  }
}
```

Uso no Componente Pai

```
<app-notifier [value]="'Novo valor!'"
(valueUpdated)="handleUpdate($event)"></app-notifier>
```

No Componente Pai (TS)

```
handleUpdate(event: string) {
  console.log(event); // Exibe: "Novo valor: Novo valor!"
}
```

Exemplo 4: Conversão de Dados no Setter

Se a entrada for um tipo complexo (como um objeto), você pode transformá-lo no setter.

```
@Component({
  selector: 'app-object-transformer',
  template: `<p>{{ transformedObject }}</p>`,
})
export class ObjectTransformerComponent {
  private _data: any;

  @Input()
  set data(value: any) {
    // Converte o objeto recebido em um JSON formatado
    this._data = JSON.stringify(value, null, 2);
  }

  get transformedObject(): string {
    return this._data;
  }
}
```

Uso no Componente Pai

```
<app-object-transformer [data]="{ name: 'Jessica', age: 28
}"></app-object-transformer>
```

Resultado na Tela

```
{
  "name": "Jessica",
  "age": 28
}
```

Resumo do Que Você Pode Fazer no Setter

1. **Transformar Dados:**
    - Exemplo: Converte o valor para maiúsculas, formata como moeda, etc.
2. **Validar Dados:**
    - Exemplo: Verifica se o valor está dentro de um intervalo aceitável.
3. **Disparar Ações ou Eventos:**
    - Exemplo: Emite um evento para informar que o valor foi alterado.
4. **Trabalhar com Dados Complexos:**
    - Exemplo: Processa ou converte objetos antes de armazená-los.
5. **Atualizar Outras Propriedades Internas:**
    - Exemplo: Baseia-se no valor de entrada para calcular outras propriedades.

Com essas abordagens, você pode adicionar uma camada extra de controle sobre os dados recebidos no componente, tornando-o mais robusto e reutilizável.