paradigmas de programação

uma visão geral sobre **orientação a objetos**
e **programação funcional**

Marcel Gonçalves dos Santos
@marcelgsantos

Marcel Gonçalves dos Santos

@marcelgsantos

desenvolvedor web full-stack

pensandonaweb.com.br

TRÊS

sp.femug.com

@femugsp

phpsp

phpsp.org.br

@phpsp

Interaja nas mídias sociais!

- fale sobre o evento, palestrantes e conteúdo
- tire fotos do evento e publique
- interaja com outros participantes do evento
- tire dúvidas ou dê feedbacks para os palestrantes

Concorra a um livro da Casa do Código!

1. seguir @marcelgsantos no Twitter
2. tuitar utilizando as hashtags #TheDevConf, #TrilhaJavaScript e #JavaScript
3. não vale tuíte em branco e retuíte
4. ler e preencher este simples formulário bit.ly/sorteio-tdc-3

O que é **paradigma de programação?**

são **modelos** ou **estilos de programação** suportados por linguagens que agrupam certas características comuns

os paradigmas de programação definem como os códigos são estruturados...

principais paradigmas de programação

os dois principais paradigmas são o imperativo e o declarativo

paradigma imperativo

descreve a resolução de um problema através de comandos que o computador pode compreender e executar

paradigma imperativo

os paradigmas procedural e orientado a objetos são exemplos de paradigmas imperativos

```
let result = 0;

// Imperative code to sum 1 to 10
for (let i = 0; i <= 10; i++) {
    result += i;
}

console.log(result); // 55
```

paradigma declarativo

permite especificar o que deve ser computado e não como deve ser computado

paradigma declarativo

os paradigmas funcional e lógico são exemplos de paradigmas declarativos

```javascript
const numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

// Declarative code to sum 1 to 10
const sum = (a, b) => a + b;
const result = numbers.reduce(sum);

console.log(result);
```

paradigmas de linguagens de programação

imperativo

procedural - C e Pascal
orientado a objetos - C++, Java, PHP, Python e Ruby

declarativo

lógico - Prolog
funcional - Clojure, Elixir, Elm, Erlang, F#, Haskell, Lisp, OCaml e Scala

Programação
Orientada a Objetos

trata da **comunicação** entre objetos através da troca de **mensagens**

um **objeto** é uma representação concreta de uma abstração...

...que possui características, comportamentos e **estado atual**

a orientação a objetos pode ser **baseada em classes** ou **baseada em protótipos**

em programação orientada a objetos baseada em classes as **classes** são definidas de antemão e **objetos** são instanciados baseados em classes

```javascript
// class to create a Person object
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    // sayName method
    sayName() {
        console.log('Hi, my name is ' + this.name + '!');
    }
}

// create an instance of Person
let person = new Person('John', 32);
person.sayName(); // Hi, my name is John!
```

em programação orientada a objetos baseada em protótipos os objetos são as **entidades primárias** e não existe nenhuma classe

```javascript
// constructor function to create a Person object
function Person(name, age) {
    this.name = name;
    this.age = age;
}

// sayName method is added to the prototype of Person
Person.prototype.sayName = function() {
    console.log('Hi, my name is ' + this.name + '!');
};

// create an instance of Person
let person = new Person('John', 32);
person.sayName(); // Hi, my name is John!
```

em JavaScript, as classes são **açúcar sintático** para funções construtoras

```javascript
// constructor functions to create a Person object
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    sayName() {
        console.log('Hi, my name is ' + this.name + '!');
    }
}

// add sayAge method to the Person prototype
Person.prototype.sayAge = function() {
    console.log('I\'m ' + this.age + ' years old!');
};

// create an instance of Person
let person = new Person('John', 32);
person.sayName(); // Hi, my name is John!
person.sayAge(); // I'm 32 years old!
```

o protótipo de um objeto é apenas **outro objeto** que o objeto é ligado

todo objeto possui uma ligação com o protótipo (e apenas uma)

novos objetos são criados baseados em **objetos já existentes** escolhidos como seu protótipo

Pilares da Orientação a Objetos

abstração

trata da representação de um objeto da vida real dentro do sistema

```javascript
// class to create a Person object
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    // sayName method
    sayName() {
        console.log('Hi, my name is ' + this.name + '!');
    }
}

// create an instance of Person
let person = new Person('John', 32);
person.sayName(); // Hi, my name is John!
```

herança

permite o reaproveitamento de código em que uma classe herda características e atributos de uma classe base

encapsulamento

permite ocultar a implementação interna de um objeto

```javascript
// hiding data through convention
class Person {
    constructor(name, age) {
        this._name = name;
        this._age = age;
    }

    sayName() {
        console.log('Hi, my name is ' + this._name + '!');
    }
}

// create an instance of Person
let person = new Person('John', 'Doe', 32);

// accessing hiding data
console.log(person._name); // John
```

```javascript
// hiding data using private fields
class Person {
    #name;
    #age;
    constructor(name, age) {
        this.#name = name;
        this.#age = age;
    }

    sayName() {
        console.log('Hi, my name is ' + this.#name + '!');
    }
}

let person = new Person('John', 'Doe', 32);

// accessing through accessor method and property
console.log(person.sayName()); // Hi, my name is John!
console.log(person.name); // undefined
console.log(person.#name); // Uncaught SyntaxError: Undefined private
field #name: must be declared in an enclosing class
```

polimorfismo

consiste na alteração do funcionamento interno de um método herdado do pai

Princípios da Orientação a Objetos

os **princípios de design** ajudam a projetar
códigos melhores

coesão

indica o grau de relação entre os membros de um módulo

```javascript
// not cohesive class
class Cart {
    constructor() {
        this._items = [];
    }

    numberOfItems() {
        return this._length;
    }

    calculateDeliveryPrice() {
        // code used to calculate the delivery price
    }
}
```

acoplamento

indica o grau de dependência entre módulos

o acoplamento ocorre quando o código de um módulo **utiliza código** de outro módulo, seja ao chamar uma função ou acessar algum dado

```
class Engine {
    start() {
        console.log('Starting the engine');
    }
}

class Car {
    constructor() {
        this.engine = new Engine;
    }

    start() {
        this.engine.start();
    }
}

// create an instance of Car
let toyota = new Car();
toyota.start(); // Starting the engine
```

ao controlar o acoplamento, o software torna-se **mais flexível** e **fácil de manter**

pode-se reduzir o acoplamento através da injeção de dependências

```
class Engine {
    start() {
        console.log('Starting the engine');
    }
}

class Car {
    constructor(engine) {
        this.engine = engine;
    }

    start() {
        this.engine.start();
    }
}

// injecting an engine dependency into the car
let engine = new Engine();
let toyota = new Car(engine);
toyota.start(); // Starting the engine
```

utilizar injeção de dependências **auxilia nos testes unitários** pois tornam os módulos fracamente acoplados, altamente coesos e facilita o mocking de objetos

"prefira classes com alta coesão e baixo acoplamento"

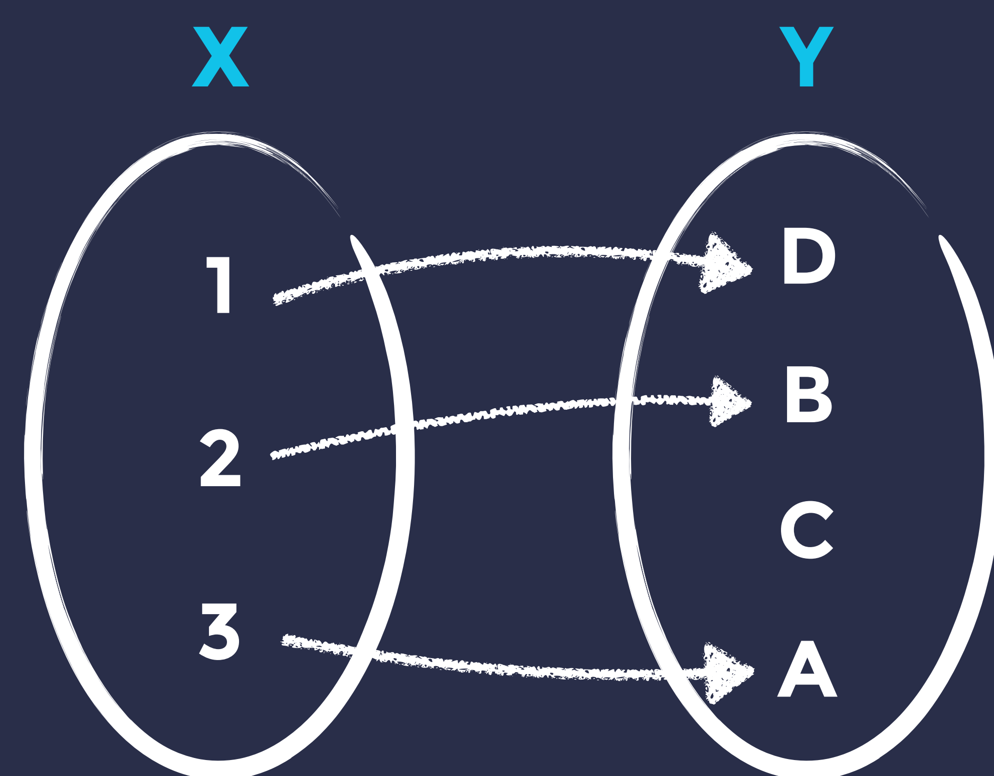
Programação **Funcional**

paradigma de programação que utiliza **funções puras** e foca na **transformação do estado**

baseado no **cálculo lambda** proposto por Alonzo Church na década de 30

na programação funcional as **funções** são tratadas como conceito principal

Uma função matemática trata-se de um simples mapeamento entre o domínio e o contra-domínio.



Estado

o estado de um programa é representado pelos valores dos dados armazenados na memória...

...em qualquer ponto de execução do programa

o estado de uma aplicação é alterado a cada interação feita pelo usuário ou pelo próprio sistema...

...e pode ser representado por uma **estrutura de dados**

a maioria dos bugs são relacionados ao
controle de estado

Funções
Puras

funções puras

1. ter parâmetros de entrada
2. não depender do estado externo
3. retorno baseado nos valores de entrada
4. não devem causar efeitos colaterais

```javascript
// pure or impure function?

let counter = 0;

function increment() {
    counter++;
    return counter;
}

console.log(increment()); // 1
```

```javascript
// pure function (ES5 syntax)

function add(x, y) {
    return x + y;
}

console.log(add(2, 3)); // 5
```

```javascript
// pure function (ES6 syntax)
const add = (x, y) => x + y;
console.log(add(2, 3)); // 5
```

por que utilizar funções puras?

são reutilizáveis, componíveis, fáceis de testar, fáceis de cachear e paralelizáveis

transparência referencial

propriedade que garante que a saída de uma função pura sempre será a mesma dado um mesmo conjunto de argumentos

```javascript
// referential transparency

const add = (x, y) => x + y;

console.log(add(2, 3) === 5); // true
console.log(5 === 5); // true
```

pode não ser fácil criar **funções puras**

porém, a **restritividade** ajuda a melhorar o foco

Mais sobre **Funções**

funções de alta ordem e
funções de primeira classse
são funções que podem ser atribuídas a variáveis, passadas como argumentos e retornadas de uma função

```javascript
// high-order function

const add = (x, y) => x + y;
const numbers = [1, 2, 3, 4, 5];

const sum = numbers.reduce(add);
const sum10 = numbers.reduce(add, 10);

console.log(sum); // 15
console.log(sum10); // 25
```

funções anônimas

funções que não possuem nome e que, geralmente, são passadas como argumento ou atribuídas

```javascript
// anonymous function
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((x, y) => x + y);
console.log(sum); // 15
```

closures

funções que possuem acesso à valores do

escopo externo

```javascript
// closure function

function greet(greeting) {
    return function (name) {
        return `${greeting} ${name}!`;
    };
}

const greet2 = greeting => name => `${greeting} ${name}!`;

console.log(greet('Hello')('Mary')); // Hello Mary!
console.log(greet2('Hello')('John')); // Hello John!
```

recursão

é quando uma função é definida em termos de si própria, ou seja, quando a função chama ela mesma

diferença entre função e procedimento (procedure)

uma função recebe um valor **e** retorna um resultado; **um procedimento é um** conjunto de comandos **executados numa ordem**

"Don't think of functions as a collection of instructions. Think of them as non-destructive operations on input `double = n => n * 2;`"

Eric Elliott, 2016.

https://twitter.com/_ericelliott/status/685172918784004097

memoize

técnica que permite que funções custosas sejam cacheadas para execuções posteriores mais rápidas

Imutabilidade

a **imutabilidade** diz que um dado não pode ser alterado após a sua criação

a **imutabilidade** permite maior confiança e evita que erros ocorram

o JavaScript não possui suporte a **dados imutáveis** de forma nativa

porém, pode-se trabalhar com dados imutáveis em JavaScript utilizando **algumas técnicas**

Currying e aplicação parcial

o **currying** é a técnica que permite transformar uma função que recebe múltiplos argumentos...

...em uma função que recebe apenas um argumento e que retorna uma função que aceita os argumentos restantes

a **aplicação parcial** é quando se executa uma função e passa apenas parte de seus argumentos

a **aplicação parcial** permite fazer a especialização de uma função mais genérica

```javascript
// specialization from a curried function
// using partial application

const greet =
    R.curry((greeting, name) => `${greeting} ${name}`);
const greetMorning = greet('Good Morning');

console.log(greetMorning('Alice')); // Good Morning Alice
```

currying e **aplicação parcial** são recursos muito utilizados em programação funcional

na programação funcional deve-se levar em consideração a **ordem dos parâmetros**

os parâmetros mais **genéricos** devem vir mais para o início e os parâmetros mais **específicos** devem vir mais para o final

o JavaScript não possui suporte nativo para **currying** como nas linguagens puramente funcionais Elm ou Haskell

Composição de **Funções**

a **composição** é o processo de combinar uma ou mais funções para criar uma nova função

```javascript
// creating a new function from others by composition

const sentence = 'estava à toa na vida o meu amor me chamou
  pra ver a banda passar cantando coisas de amor ';
const wordCount = R.length(R.split(' ', sentence));

console.log(wordCount); // 19
```

é uma solução **elegante** e **legível** e ajuda a evitar a utilização do aninhamento de funções

o Ramda possui uma função que permite criar uma **nova função** a partir da composição de funções

```
// create a function using composition

const sentence = 'estava à toa na vida o meu amor me chamou
  pra ver a banda passar cantando coisas de amor ';
const countWords = R.compose(R.length, R.split);

console.log(countWords(' ', sentence)); // 19
```

Biblioteca
Ramda

uma **biblioteca** construída para o estilo de programação funcional que facilita a utilização de pipelines e dados imutáveis

possui foco no estilo **puramente funcional**

todas as funções do Ramda são **auto-curried**

os argumentos das funções do Ramda são organizados de forma a facilitar a utilização de **currying**

Caso de Uso 1

somar os preços dos produtos
de um carrinho de compras

Passo 1

```
// shopping cart
const cart = [
    {id: 1, product: 'iPhone', price: 499},
    {id: 2, product: 'Kindle', price: 179},
    {id: 3, product: 'Macbook Pro', price: 1199},
];

// get prices from shopping cart and sum them
// using intermediate values
const cartPrices = R.map(item ⇒ item.price, cart);
const cartSum = R.sum(cartPrices);

console.log(cartSum); // 1877
```

faz a somatória da lista de números e retorna o total

realiza o mapeamento da lista de produtos (objetos) para uma lista de preços (números)

Passo 2

```javascript
// shopping cart
const cart = [
    {id: 1, product: 'iPhone', price: 499},
    {id: 2, product: 'Kindle', price: 179},
    {id: 3, product: 'Macbook Pro', price: 1199},
];

// get prices from shopping cart and sum them
// using function composition
const totalCart = R.compose(
    R.sum,
    R.map(item => item.price),
);

console.log(totalCart(cart)); // 1877
```

cria uma nova função a partir da composição de funções e elimina valores intermediários

a composição é feita da direita para a esquerda

aplicação parcial da função map

Passo 3

```javascript
// shopping cart
const cart = [
    {id: 1, product: 'iPhone', price: 499},
    {id: 2, product: 'Kindle', price: 179},
    {id: 3, product: 'Macbook Pro', price: 1199},
];

// get prices from shopping cart and sum them
// using function composition with pipe
const totalCart = R.pipe(
    R.map(item => item.price),
    R.sum,
);

console.log(totalCart(cart)); // 1877
```

o pipe de funções é feito da esquerda para a direita e facilita a leitura do código

Caso de Uso 2

limpar dados vindo de um formulário e realizar um cálculo

Passo 1

```
// cleaning data from an input
const price = '100';

const discount = (perc, value) => perc * value;

let priceInt = parseInt(price);
let priceDiscount = discount(0.2, priceInt);

console.log(priceDiscount); // 20
```

Passo 2

```
// cleaning data from an input
const price = '100';

const discount = (perc, value) => perc * value;

// using partial application
const discount20 = R.partial(discount, [0.2]);

let priceInt = parseInt(price);
let priceDiscount = discount20(priceInt);

console.log(priceDiscount); // 20
```

cria uma nova função a partir da aplicação parcial de uma existente

Passo 3

```
// cleaning data from an input
const price = '100';

const discount = (perc, value) => perc * value;

// using function composition
const priceDiscount = R.pipe(
    parseInt,
    R.partial(discount, [0.2]),
);

console.log(priceDiscount(price)); // 20
```

cria uma nova função a partir da composição de funções utilizando a função pipe e elimina valores intermediários

Passo 4

```
// cleaning data from an input
const price = 'lambda!';

const discount = (perc, value) => perc * value;

// using function composition
const priceDiscount = R.pipe(
    parseInt,
    R.partial(discount, [0.2]),
);

console.log(priceDiscount(price)); // null
```

erro ao receber um valor
não numérico

Passo 5

```
// cleaning data from an input
const price = 'lambda!';

const discount = (perc, value) => perc * value;

// using function composition
const priceDiscount = R.pipe(
    parseInt,
    R.defaultTo(0),
    R.partial(discount, [0.2]),
);

console.log(priceDiscount(price)); // 0
```

retorna o valor padrão para o caso de um valor não truthy

existem inúmeros conceitos relacionados a **programação funcional** como functors, monads, lazy evaluation, tail call optimization...

Conclusão

os **princípios de design** ajudam a projetar
códigos melhores

um código mau projetado é um código difícil de mudar

prefira módulos com **alta coesão** e **baixo acoplamento**

a **programação funcional** não é sobre não ter estado...

...e sim sobre **eliminar** estado e efeito colateral sempre que possível e **controlar** efeitos colaterais quando necessário

foque na **transformação do estado** e evite **efeitos colaterais**

conhecer bem os **paradigmas de programação** te permite escolher a melhor ferramenta para cada problema

vá em frente e **divirta-se**!

Referências

bit.ly/referencias-palestra-paradigmas

Avalie!

bit.ly/avalie-palestra-paradigmas

Obrigado.

Perguntas?

@marcelgsantos

speakerdeck.com/marcelgsantos