O que são Funções Puras?

Uma **função pura** é uma função que, dado um mesmo conjunto de entradas, sempre produz a mesma saída e não causa efeitos colaterais. Vamos analisar cada uma dessas características:

1. **Determinismo**:
    - Uma função pura sempre retorna o mesmo resultado para os mesmos parâmetros. Por exemplo, se você tem uma função que soma dois números, como `soma(a, b)`, sempre que você chamar `soma(2, 3)`, o resultado será `5`. Isso é previsível e fácil de entender.
2. **Sem Efeitos Colaterais**:
    - Funções puras não alteram nenhum estado externo. Isso significa que elas não modificam variáveis fora de seu escopo, não escrevem em arquivos, não alteram bancos de dados e não interagem com o mundo exterior. Continuando com o exemplo da soma, se `soma(a, b)` apenas retorna `a + b` e não altera nada fora dela, é uma função pura.

Vantagens das Funções Puras

1. **Testabilidade**:
    - Funções puras são mais fáceis de testar, já que você só precisa verificar a entrada e a saída. Não se preocupe com o estado global ou outros efeitos.
2. **Composição**:
    - Elas podem ser facilmente compostas para formar funções mais complexas. Se você tem duas funções puras, `f` e `g`, você pode criar uma nova função `h(x) = f(g(x))` sem problemas.
3. **Paralelismo**:
    - Como não há efeitos colaterais, é mais seguro executar funções puras em paralelo, pois não há risco de uma função interferir na execução de outra.

Funções puras são fundamentais em programação funcional e trazem muitos benefícios em termos de previsibilidade, testabilidade e facilidade de composição. Ao se concentrar em criar funções puras, você pode escrever um código mais limpo e gerenciável.

Exemplos:

```
function soma(a, b) {
   return a + b;
}

// Testando a função
console.log(soma(2, 3)); // 5
console.log(soma(2, 3)); // 5 (mesmo resultado sempre)
```

```
function quadrado(x) {
   return x * x;
}

// Testando a função
console.log(quadrado(4)); // 16
console.log(quadrado(4)); // 16 (sempre o mesmo resultado)
```

Aplicação:

Cenário: Calculadora de Preço de Produtos

Vamos construir uma pequena aplicação que calcula o preço total de produtos em um carrinho de compras.

Estrutura do Projeto

1. **Funções Puramente Funcionais**:
    - `calcularPrecoTotal(produtos)`: calcula o preço total sem alterar o estado global.
    - `adicionarProduto(produtos, produto)`: adiciona um produto ao carrinho.
2. **Funções Impuras**:
    - `atualizarCarrinho(produto)`: modifica uma variável global que representa o carrinho.

```
*function calcularPrecoTotal(produtos) {
    return produtos.reduce((total, produto) => total +
produto.preco, 0);
}

function adicionarProduto(produtos, produto) {
    return [...produtos, produto];
}

// Exemplo de uso
const produtos = [
    { nome: "Produto A", preco: 50 },
    { nome: "Produto B", preco: 30 }
];

const novoProduto = { nome: "Produto C", preco: 20 };
```

```
const novoCarrinho = adicionarProduto(produtos, novoProduto);

console.log(calcularPrecoTotal(novoCarrinho)); // 100
```

O que são funções de alta ordem?

Funções de alta ordem são aquelas que podem receber outras funções como argumentos ou retornar uma função como resultado. Esse conceito é comum em linguagens de programação funcional, como JavaScript, Python, Haskell, entre outras.

Características principais:

1. **Receber funções como argumentos**: Você pode passar uma função para outra função.
2. **Retornar funções**: Uma função pode retornar outra função.

Vantagens das Funções de Alta Ordem

1. **Reuso de Código**: Elas permitem que você reutilize lógica sem duplicar código.
2. **Composição**: Você pode compor funções de maneira mais fácil, criando funções complexas a partir de funções simples.
3. **Legibilidade**: O uso de funções de alta ordem pode tornar seu código mais expressivo e fácil de entender.

Aplicações Comuns

- **Map**: Uma função de alta ordem que aplica uma função a cada elemento de um array.
- **Filter**: Filtra elementos de um array com base em uma condição.
- **Reduce**: Reduz um array a um único valor, aplicando uma função a cada elemento.

Funções de alta ordem são uma ferramenta poderosa em programação que permite maior flexibilidade e expressividade no seu código. Elas ajudam a estruturar seu programa de maneira modular e reutilizável, tornando a manipulação de dados e a criação de lógica mais eficiente

Exemplos:

1. Função que recebe outra função como argumento

```
function aplicarOperacao(array, operacao) {
    let resultado = [ ];
    for (let i = 0; i < array.length; i++) {
        resultado.push(operacao(array [i] ));
    }
    return resultado;
}
function dobro(x) {
    return x * 2;
}
const numeros = [1, 2, 3, 4];
const numerosDobrado = aplicarOperacao(numeros, dobro);
console.log(numerosDobrado); // [2, 4, 6, 8]
```

Nesse exemplo:

- `aplicarOperacao` é uma função de alta ordem que aceita uma função (`operacao`) como argumento.
- `dobro` é a função que define a operação que queremos aplicar.

2. Função que retorna outra função

```
function criarMultiplicador(fator) {
    return function(numero) {
        return numero * fator;
    };
}
const multiplicarPor3 = criarMultiplicador(3);
console.log(multiplicarPor3(5)); // 15
```

Nesse exemplo:

`criarMultiplicador` é uma função que retorna uma nova função.

A nova função multiplicará qualquer número pelo `fator` fornecido quando a função é chamada.

Aplicação:

Cenário

Suponha que temos um array de objetos representando produtos, onde cada produto tem um nome, uma categoria e um preço. Vamos utilizar funções de alta ordem para:

1. Filtrar produtos por categoria.
2. Aplicar um desconto a esses produtos.
3. Calcular o preço total dos produtos filtrados e descontados.

```
// Array de produtos
const produtos = [
    { nome: "Camiseta", categoria: "Roupas", preco: 50 },
    { nome: "Calça", categoria: "Roupas", preco: 80 },
    { nome: "Tênis", categoria: "Calçados", preco: 120 },
    { nome: "Relógio", categoria: "Acessórios", preco: 300 },
    { nome: "Boné", categoria: "Acessórios", preco: 40 }
];
// Função para filtrar produtos por categoria
function filtrarPorCategoria(produtos, categoria) {
    return produtos.filter(produto => produto.categoria === categoria);
}
// Função para aplicar desconto
function aplicarDesconto(produtos, desconto) {
    return produtos.map(produto => {
        return { ...produto, preco: produto.preco * (1 - desconto) };
```

```
    });
}
// Função para calcular o preço total
function calcularPrecoTotal(produtos) {
    return produtos.reduce((total, produto) => total + produto.preco, 0);
}
// Aplicação do conceito
const categoriaEscolhida = "Acessórios";
const desconto = 0.10; // 10%
// Filtra produtos pela categoria
const produtosFiltrados = filtrarPorCategoria(produtos, categoriaEscolhida);
console.log("Produtos filtrados:", produtosFiltrados);
// Aplica desconto nos produtos filtrados
const produtosComDesconto = aplicarDesconto(produtosFiltrados, desconto);
console.log("Produtos com desconto:", produtosComDesconto);
// Calcula o preço total
const precoTotal = calcularPrecoTotal(produtosComDesconto);
console.log("Preço total após desconto:", precoTotal.toFixed(2));
```

Array de produtos: Contém objetos que representam os produtos, cada um com nome, categoria e preço.

Filtrar por categoria:

A função `filtrarPorCategoria` usa `filter` para retornar apenas os produtos que pertencem à categoria especificada.

Aplicar desconto:

A função `aplicarDesconto` utiliza `map` para criar um novo array, onde o preço de cada produto é ajustado de acordo com o desconto.

Calcular preço total:

A função `calcularPrecoTotal` utiliza `reduce` para somar os preços de todos os produtos, retornando o total.

O que é Currying?

Currying é uma técnica de programação funcional que transforma uma função que aceita múltiplos argumentos em uma sequência de funções que cada uma aceita um único argumento. Em vez de chamar a função com todos os seus argumentos de uma vez, você a chama com um argumento, e ela retorna outra função que espera o próximo argumento.

Por que usar Currying?

1. **Facilidade de Reuso**: Você pode criar funções específicas a partir de funções mais gerais.
2. **Criação de Funções Parciais**: Permite fixar alguns argumentos de uma função, criando uma nova função com menos parâmetros.
3. **Melhor Leitura**: Em algumas situações, o código pode se tornar mais legível.

Exemplo:

Função Normal

Imaginemos uma função que soma três números**:**

```
function soma(a, b, c) {
    return a + b + c;
}
```

Para usar essa função, precisaríamos passar todos os três argumentos:

```
console.log(soma(1, 2, 3)); // Saída: 6
```

Agora, vamos transformar essa função em uma versão curried:

```
function somaCurried(a) {
```

```
  return function(b) {
      return function(c) {
          return a + b + c;
      }}}
```

Aqui, `somaCurried` aceita o primeiro argumento `a` e retorna uma nova função que aceita `b`, que por sua vez retorna outra função que aceita `c`

O *currying* é uma técnica poderosa na programação funcional, permitindo a construção de funções mais flexíveis e reutilizáveis. Com a prática, você poderá utilizá-la em suas aplicações para tornar seu código mais limpo e eficiente.

Aplicação

Cenário

Suponha que você queira calcular o total de um pedido em um restaurante, onde os preços dos itens variam e você pode ter descontos ou taxas de serviço.

Passo 1: Criar uma Função Curried

Vamos criar uma função que aceita o preço do item, o desconto e a taxa de serviço. Usaremos currying para permitir que a função seja chamada em etapas.

```
function calcularTotal(preco) {
    return function(desconto) {
        return function(taxaServico) {
            const total = preco - desconto + taxaServico;
            return total.toFixed(2); // Formata para duas casas decimais
        }}}
```

Passo 2: Usar a Função Curried

Agora vamos usar nossa função curried para calcular o total de um pedido.

```
// Vamos calcular o total de um item com preço 50
const precoItem = 50;


// Criar uma função que fixa o preço do item
```

```
const totalComPreco = calcularTotal(precoItem);

// Agora, podemos aplicar um desconto de 10

const totalComDesconto = totalComPreco(10);

// E, por fim, adicionar uma taxa de serviço de 5

const totalFinal = totalComDesconto(5);

console.log(`Total do pedido: R$ ${totalFinal}`); // Saída: Total do pedido: R$ 45.00
```

Passo 3: Usar de Forma Mais Compacta

Com a função curried, também podemos fazer isso de forma mais compacta:

```
const totalPedido = calcularTotal(50)(10)(5);

console.log(`Total do pedido: R$ ${totalPedido}`); // Saída: Total do pedido: R$ 45.00
```

O que é Composição de Funções?

A composição de funções é um conceito matemático que também é muito utilizado na programação. Em termos simples, a composição de funções envolve a aplicação de uma função dentro de outra. Se temos duas funções, f(x)f(x)f(x) e g(x)g(x)g(x), a composição delas é denotada como (f∘g)(x)(f \circ g)(x)(f∘g)(x), que significa que primeiro aplicamos ggg e depois aplicamos fff ao resultado de ggg.

Como Funciona na Programação?

Em programação, a composição de funções permite que você crie funções mais complexas a partir de funções mais simples

Vantagens da Composição de Funções

1. **Reusabilidade**: Funções menores e bem definidas podem ser reutilizadas em diferentes composições.
2. **Clareza**: Compor funções pode tornar o código mais legível, já que você pode descrever operações complexas em termos de operações mais simples.
3. **Manutenção**: Se precisar mudar o comportamento de uma parte da lógica, você pode modificar apenas uma função, sem afetar as outras.

Exemplos:

Compondo as Funções

```
function compose(f, g) {
return function(x) {
      return f(g(x));
  }}
const doubleThenAddThree = compose(addThree, double);
```

Neste caso, estamos dizendo que queremos primeiro dobrar o número e depois adicionar 3.

Usando a Função Composta

```
const result = doubleThenAddThree(4); // Isso vai ser addThree(double(4))
console.log(result); // Saída: 11
```

Passo 1: `double(4)` retorna `8`.

Passo 2: `addThree(8)` retorna `11`.

Aplicação:

Cenário: Processamento de Preços de Produtos

Imagine que você tem um conjunto de produtos e precisa calcular o preço final após aplicar um desconto e, em seguida, adicionar impostos.

1. Definindo as Funções

Função para aplicar desconto (ex: 10%):

```
function applyDiscount(price) {
   return price * 0.9; // 10% de desconto
}
```

Função para adicionar imposto (ex: 15%):

```
function addTax(price) {
```

```
    return price * 1.15; // 15% de imposto
}
```

2. Compondo as Funções

Vamos criar uma função que primeiro aplica o desconto e, em seguida, adiciona o imposto:

```
function compose(f, g) {
    return function(x) {
        return f(g(x));
    }}
const finalPrice = compose(addTax, applyDiscount);
```

3. Usando a Função Composta

Agora, vamos calcular o preço final de um produto que custa $100:

```
const originalPrice = 100;
const result = finalPrice(originalPrice);
console.log(`O preço final é: $${result.toFixed(2)}`); // Saída: O preço final é: $103.50
```

Explicação do Resultado

Passo 1: `applyDiscount(100)` retorna `90` (aplicando 10% de desconto).

Passo 2: `addTax(90)` retorna `103.5` (adicionando 15% de imposto)