

UNIVERSIDADE DE FORTALEZA

CURSO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

DISCIPLINA: PROGRAMAÇÃO FUNCIONAL

ATIVIDADE PARCIAL:

Conversor de Moedas Baseado em Programação Funcional com JavaScript

EQUIPE:

Alanna Mércia de França Vidal - 2313687

Caio Ruan Carvalho Vasconcelos – 2315923

Jonas Tiago dos Santos - 2313620

Laiza Silva Fernandes – 2317162

Lizandra Raquel da Silva Assunção - 2319725

Maria Lindayse Silva - 2314012

FORTALEZA-CE

MARÇO /2024

Documento de Requisitos

1. Introdução

Este documento define os requisitos para o Conversor de Moedas, um projeto desenvolvido com base nos conceitos de Programação Funcional. O sistema incorpora o uso de funções lambda, closures, funções de alta ordem e list comprehensions, promovendo uma abordagem funcional e eficiente.

O programa solicita ao usuário informações como a moeda inicial, a moeda final e o valor a ser convertido. Por meio de uma lista numerada de moedas, apresentada de maneira clara, o usuário pode selecionar as moedas desejadas inserindo apenas o número correspondente, o que simplifica e agiliza a interação. Utilizando taxas de câmbio pré-definidas, o sistema realiza os cálculos necessários e exibe o resultado da conversão de forma precisa e bem formatada.

Além disso, o programa se destaca por incluir um recurso prático de histórico, que armazena as duas últimas conversões realizadas. Essa funcionalidade permite que o usuário acompanhe facilmente suas operações mais recentes. A qualquer momento, é possível optar por realizar uma nova conversão ou consultar o histórico, proporcionando uma experiência funcional e acessível.

Objetivo

Construir um conversor de moedas que:

Obtenha e valide dados do usuário.

Consuma uma API externa para recuperar a taxa de câmbio.

Converta valores com precisão utilizando funções de alta ordem e lambda.

Mantenha um histórico das 2 últimas conversões por meio de uma closure.

Exiba de forma clara os resultados e o histórico de conversões.

2. Definição de Papéis:

CATEGORIA	DETALHES	RESPONSÁVEL
Documentação de requisitos	Criação do documento, atribuição de papéis, de possíveis manutenções no arquivo do projeto e revisão do documento conferindo se todos os requisitos estão presentes.	Lizandra Raquel da Silva Assunção Maria Lindayse Silva
Aplicação da função lambda	Implementação de uma função lambda para formatar o valor final da conversão, ajustando casas decimais e símbolos monetários.	Alanna Mércia de França Vidal
Aplicação da função list comprehension.	Simulação do comportamento de list comprehension utilizando métodos como map para numerar as moedas e exibir uma lista numerada de forma clara para o usuário.	Laiza Silva Fernandes
Aplicação da função closure.	Adicionar a função closure para armazenar o histórico das 02 últimas conversões de moeda, mas com o registro limitado das mais recentes, descartando as mais antigas à medida que novas conversões são adicionadas.	Lizandra Raquel da Silva Assunção
Aplicação da função de alta ordem.	Uso de funções de alta ordem para obter as taxas de câmbio e abstrair a lógica de cálculo.	Jonas Tiago dos Santos
Casos de teste	Implementação de casos de teste para verificar o funcionamento e validade do programa.	Caio Ruan Carvalho Vasconcelos

3. Utilização de Conceitos de Programação Funcional:

- **Função Lambda:** é uma função definida em uma única linha, geralmente contendo uma lógica relativamente simples. **implementação:** Foi utilizada uma função lambda para formatar o valor convertido e exibir o resultado da conversão, incluindo casas decimais e as moedas envolvidas. **Código:**

```
// (Função lambda) - Alanna Mércia
const formatarResultado = (
  valor,
  moedaInicial,
  valorConvertido,
  moedaFinal
) =>
  `Resultado: ${valor.toFixed(
    2
  )} ${moedaInicial} equivale a ${valorConvertido.toFixed(2)} ${moedaFinal}`;

return formatarResultado(valor, moedaInicial, valorConvertido, moedaFinal);
}
```

- **List Comprehension:** JavaScript não tem suporte nativo para List Comprehension, o comportamento foi simulado. **Implementação:** ele permite transformar listas de forma concisa e legível. Para implementar utilizamos (map) para numerar as moedas e exibir uma lista numerada. **código:**

```
//Adicionar a list comprehension - Laiza S. Fernandes
//Deve listar as moedas numeradas e solicitar que o usuário escolhao
// núm correspondente a moeda que deseja converter

function listarMoedas() {
  const moedas = ['EURO', 'DOLAR', 'REAL', 'LIBRA'];
  const listaFormatada = moedas.map((moeda, index) => `${index + 1} - ${moeda}`);
  console.log('Moedas disponíveis:\n' + listaFormatada.join('\n'));

  //Loop que solicita a moeda e verifica se a escolha é válida
  let escolha;
  do {
    escolha = parseInt(prompt('Escolha o número da moeda: '));

    if (isNaN(escolha)) {
      console.log('Por favor, insira um número válido.');
```

- **Closure:** Função que mantém o estado de variáveis privadas entre execuções. Utilizamos para criar funções com estado interno privado, acessível apenas pelas funções retornadas. **Implementação:** no projeto de conversor de moeda, a closure foi usada para gerenciar o histórico de conversões, armazenando somente as duas últimas e garantindo acesso controlado ao histórico. **Código:**

```
//Lizandra Raquel da Silva Assunção
//criação da função closure para exibir o historico das 2 ultimas conversões.
//Ao realizar mais de 2 conversões
//A primeira conversão feita é removida dando lugar a uma nova conversão.
```

```
const criarHistoricoConversoes = () => {
  let historico = [];
  const adicionarConversao = (resultado) => {
    historico.unshift(resultado);
    if (historico.length > 2) {
      historico.pop();
    }
  };
  const mostrarHistorico = () => historico;
  return { adicionarConversao, mostrarHistorico };
};
const historicoConversoes = criarHistoricoConversoes();
const obterHistorico = () => historicoConversoes.mostrarHistorico();

const resultado = iniciar_programa_conversao(); console.log(resultado)
```

- **Alta ordem:** Uma função que aceita outra função como argumento ou retorna outra função. **Implementação:** No projeto, a função obter_taxa_cambio retorna uma função que calcula a taxa de câmbio com base nas moedas fornecidas. **Código:**

```
//Função de alta ordem, para obter a taxa de câmbio - jonas
const obterTaxa = obter_taxa_cambio(moedaInicial, moedaFinal);
const taxa = obterTaxa();

if (taxa === null) {
  console.log('Conversão não disponível para essas moedas.');
```

```
  return;
}

const valorConvertido = valor * taxa;
```

4. Requisitos Funcionais:

REQUISITO	DESCRIÇÃO	CÓDIGO
coleta de dados do usuário	O programa deve solicitar moedas de entrada e saída e o valor a ser convertido.	coletar_dados_usuario
Conversão de moeda	O programa deve calcular a conversão com base nas taxas de câmbio definidas.	converter_moeda, que utiliza a função de alta ordem obter_taxa_cambio para obter a taxa.
Listagem de Moedas com List Comprehensions	O programa deve exibir uma lista numerada das moedas disponíveis para conversão, permitindo que o usuário selecione uma moeda inicial e final inserindo apenas o número correspondente.	Implementado na função listarMoedas utilizando o método map.
Validação dos dados inseridos pelo usuário	O sistema deve validar se as moedas escolhidas estão dentro das opções disponíveis e se o valor inserido é um número válido e maior que zero.	coletar_dados_usuario com uso de loops do...while
Formato do resultado	O programa deve exibir o resultado da conversão em formato claro.	Implementado na função lambda formatarResultado dentro de converter_moeda
Armazenamento do Histórico de Conversões	O programa deve armazenar as duas últimas conversões realizadas.	Implementado utilizando a closure criarHistoricoConversoes
Visualização do Histórico	O programa deve permitir visualizar o histórico ou realizar uma nova conversão.	Implementado no loop principal de iniciar_programa_conversao com base na entrada do usuário

5. Requisitos Não Funcionais:

REQUISITO	DESCRIÇÃO	CÓDIGO
Execução em Ambiente Node.js	O sistema deve ser executado em um ambiente que suporte Node.js	Uso do módulo prompt-sync para interagir com o usuário no terminal.
Desempenho em Múltiplas Conversões	O sistema deve ser capaz de processar várias conversões consecutivas sem reiniciar.	Estrutura de repetição while (true) no loop principal de iniciar_programa_conversao.
Modularidade do Código	O código deve ser organizado em funções separadas para facilitar manutenção e extensibilidade.	Implementação modular com funções como coletar_dados_usuario, converter_moeda, obter_taxa_cambio, e criarHistoricoConversoes.
Taxas de Câmbio Mockadas	O sistema deve usar taxas de câmbio mockadas armazenadas localmente.	Definido como um objeto na função obter_taxa_cambio.
Limitação de Histórico	O histórico de conversões deve armazenar no máximo duas conversões, removendo as mais antigas automaticamente.	Implementado na closure criarHistoricoConversoes.
Mensagens de Erro e Repetição de Entradas Inválidas	O sistema deve exibir mensagens de erro claras e solicitar novamente os dados em caso de entradas inválidas.	Tratamento de erros e mensagens nas funções coletar_dados_usuario e converter_moeda

6. Casos de teste:

Descrição	Resultado Esperado
Verificar se o programa coleta dados corretamente.	<p>Valores válidos de moeda inicial, moeda final e valor numérico.</p> <pre>@CaioRuanCarv → /workspaces/programacao-ofuncional (main) \$ node index.js Programa de conversão de moedas Escolha a moeda inicial: Moedas disponíveis: 1 - EURO 2 - DOLAR 3 - REAL 4 - LIBRA Escolha o número da moeda: 3 Escolha a moeda final: Moedas disponíveis: 1 - EURO 2 - DOLAR 3 - REAL 4 - LIBRA Escolha o número da moeda: 2 Digite o valor a ser convertido: 100 Resultado da Conversão: Resultado: 100.00 REAL equivale a 20.00 DOLAR Digite "nova" para fazer outra conversão ou "histórico" para ver o histórico: █</pre>
Testar a conversão entre diferentes pares de moedas.	<p>O valor convertido deve corresponder às taxas mockadas.</p> <pre>@CaioRuanCarv → /workspaces/programacao-ofuncional (main) \$ node index.js Programa de conversão de moedas Escolha a moeda inicial: Moedas disponíveis: 1 - EURO 2 - DOLAR 3 - REAL 4 - LIBRA Escolha o número da moeda: 3 Escolha a moeda final: Moedas disponíveis: 1 - EURO 2 - DOLAR 3 - REAL 4 - LIBRA Escolha o número da moeda: 2 Digite o valor a ser convertido: 100 Escolha o número da moeda: 3 Escolha a moeda final: Moedas disponíveis: 1 - EURO 2 - DOLAR 3 - REAL 4 - LIBRA Escolha o número da moeda: 4 Digite o valor a ser convertido: 2567.8 Resultado da Conversão: Resultado: 2567.80 REAL equivale a 385.17 LIBRA Digite "nova" para fazer outra conversão ou "histórico" para ver o histórico: █ Digite "nova" para fazer outra conversão ou "histórico" para ver o histórico: nova Escolha a moeda inicial: Moedas disponíveis: 1 - EURO 2 - DOLAR 3 - REAL 4 - LIBRA Escolha o número da moeda: 2 Escolha a moeda final: Moedas disponíveis: 1 - EURO 2 - DOLAR 3 - REAL 4 - LIBRA Escolha o número da moeda: 1 Digite o valor a ser convertido: 789.95 Resultado da Conversão: Resultado: 789.95 DOLAR equivale a 734.65 EURO</pre>
Validar o histórico de conversões.	<p>O histórico deve armazenar apenas as duas últimas conversões.</p> <pre>Digite "nova" para fazer outra conversão ou "histórico" para ver o histórico: historico Histórico: Resultado: 789.95 DOLAR equivale a 734.65 EURO Resultado: 2567.80 REAL equivale a 385.17 LIBRA</pre>
Testar entradas inválidas do usuário.	<p>O programa deve exibir mensagens de erro e repetir a solicitação.</p> <pre>Digite "nova" para fazer outra conversão ou "histórico" para ver o histórico: a Opção inválida. Tente novamente. Digite "nova" para fazer outra conversão ou "histórico" para ver o histórico: █</pre>

Conclusão dos testes realizados:

Durante os testes realizados no programa, os seguintes resultados foram observados. Inicialmente, foi testada a coleta de dados, onde foram inseridos valores válidos para a moeda inicial, moeda final e o valor numérico. O programa demonstrou funcionar corretamente, associando as entradas do usuário às moedas e valores fornecidos, sem qualquer problema. Em seguida, testou-se a conversão entre diferentes pares de moedas, como por exemplo, de EURO para REAL e de DOLAR para LIBRA. Os cálculos resultaram nos valores esperados, correspondendo precisamente às taxas de câmbio mockadas no programa. A funcionalidade de gerenciamento do histórico de conversões foi avaliada realizando múltiplas operações consecutivas. O programa manteve corretamente somente as duas últimas conversões, descartando as mais antigas, como especificado. Por fim, foram testadas entradas inválidas, incluindo moedas inexistentes, valores negativos ou caracteres não numéricos. O programa reagiu conforme esperado, exibindo mensagens de erro apropriadas e solicitando novas entradas até que informações válidas fossem fornecidas. De forma geral, todos os casos de teste foram bem-sucedidos.

7. Ferramentas utilizadas:

Usamos a linguagem JavaScript para criação das funções
Usamos o Vscode para criação do trabalho.

8: GitHub:

<https://github.com/alanna-mercia/programacao-ofuncional>

9: Adição do Uso do Chatbot e Recursos Externos

Durante o desenvolvimento do projeto, utilizamos um chatbot para auxiliar na pesquisa de conceitos e implementação prática, como um complemento para os materiais de estudos disponíveis. Também ajudou a equipe a entender como implementar esses conceitos em JavaScript e forneceu exemplos práticos. A seguir, detalhamos como o chatbot foi usado e os materiais complementares encontrados.

Pergunta: *"Conceitos de programação funcional"* **Resposta do chat:** A programação funcional é baseada em funções puras, imutabilidade e ausência de efeitos colaterais. Outros conceitos fundamentais incluem funções de ordem superior, composição de funções, avaliação, referência transparente, recursão e evitar estados globais. Linguagens como Haskell, Scala e bibliotecas JavaScript utilizam esses princípios.

Pergunta: *"Dicas de vídeos sobre programação funcional"* **Resposta do chat:** Sugestões incluem:

- *"Introdução à Programação Funcional"*: Explica os conceitos principais de maneira prática e clara.
- *"Programação funcional com JavaScript"* do canal Dev Soutinho: Demonstra técnicas funcionais no dia a dia do desenvolvimento.
- *"Como entender currying e composição de funções"*: Uma abordagem detalhada e simplificada.
- *"Imutabilidade em JavaScript (outras linguagens também)"*: Ensinando como usar imutabilidade no desenvolvimento funcional.

Pergunta: *"Como implementar programação funcional em JavaScript?"* **Resposta do chat:** Para implementar programação funcional em JavaScript, é possível usar funções puras, imutabilidade e funções de alta ordem como `map`, `filter` e `reduce`. Métodos declarativos são preferidos sobre loops tradicionais. Composição de funções, currying e a substituição de estados mutáveis por estados imutáveis são técnicas importantes.

Pergunta: *"É proibido usar loop?"* **Resposta do chat:** Não é proibido usar loops em programação funcional, mas métodos como `map`, `filter` e `reduce` são preferidos por alinharem-se melhor ao paradigma. O uso de loops é aceitável em contextos específicos onde fornecem mais controle ou clareza.

Pergunta: *"Posso criar um código com loops e também utilizar map?"* **Resposta do chat:** É possível combinar loops e métodos como `map` em um mesmo código. Um exemplo seria usar um loop para filtrar números maiores que cinco e aplicar `map` para transformá-los.