30 Exercícios Práticos sobre Funções em JavaScript

Esta lista contém 30 exercícios práticos sobre funções em JavaScript, com foco especial em funções anônimas. Os exercícios estão organizados em ordem crescente de dificuldade.

Nível Iniciante

Exercício 1: Declaração Básica de Função

Crie uma função chamada `saudacao` que não recebe parâmetros e exibe "Olá, mundo!" no console.

```javascript
// Solução
function saudacao() {
  console.log("Olá, mundo!");
}

saudacao();
```

Exercício 2: Função com Parâmetro

Crie uma função chamada `saudar` que recebe um nome como parâmetro e exibe "Olá, [nome]!" no console.

```javascript
// Solução
function saudar(nome) {
  console.log(`Olá, ${nome}!`);
}

saudar("Maria");
```

Exercício 3: Função Anônima Básica

Crie uma função anônima e atribua-a a uma variável chamada `mostrarMensagem`. A função deve exibir "Esta é uma função anônima" no console.

```javascript
// Solução
const mostrarMensagem = function() {
  console.log("Esta é uma função anônima");
};

mostrarMensagem();
```

Exercício 4: Arrow Function Simples

Converta a função anônima do exercício anterior para uma arrow function.

javascript

```javascript
// Solução
const mostrarMensagem = () => {
  console.log("Esta é uma arrow function");
};

mostrarMensagem();
```

Exercício 5: Arrow Function com Parâmetro

Crie uma arrow function que receba um número como parâmetro e retorne seu dobro.

javascript

```javascript
// Solução
const dobrar = (numero) => {
  return numero * 2;
};

console.log(dobrar(5)); // 10
```

Exercício 6: Arrow Function com Retorno Implícito

Simplifique a arrow function do exercício anterior usando retorno implícito.

javascript

```javascript
// Solução
const dobrar = numero => numero * 2;

console.log(dobrar(5)); // 10
```

Exercício 7: Função como Argumento

Crie uma função chamada `executar` que aceita uma função como argumento e a executa.

```javascript
// Solução
function executar(funcao) {
  funcao();
}

executar(() => {
  console.log("Função executada como argumento");
});
```

Exercício 8: Função que Retorna Função

Crie uma função `criarSaudacao` que recebe um parâmetro `saudacao` e retorna uma função que aceita um nome e retorna a saudação completa.

```javascript
// Solução
function criarSaudacao(saudacao) {
  return function(nome) {
    return `${saudacao}, ${nome}!`;
  };
}

const bomDia = criarSaudacao("Bom dia");
console.log(bomDia("João")); // "Bom dia, João!"
```

Exercício 9: Função Anônima com `forEach`

Crie um array de números e use `forEach` com uma função anônima para exibir cada número multiplicado por 2.

```javascript
// Solução
const numeros = [1, 2, 3, 4, 5];

numeros.forEach(function(numero) {
  console.log(numero * 2);
});
```

Exercício 10: Arrow Function com `map`

Use o método `map` com uma arrow function para criar um novo array onde cada número do array original é elevado ao quadrado.

```javascript
// Solução
const numeros = [1, 2, 3, 4, 5];
const quadrados = numeros.map(numero => numero * numero);

console.log(quadrados); // [1, 4, 9, 16, 25]
```

Nível Intermediário

Exercício 11: Função com Valor Padrão

Crie uma arrow function chamada `potencia` que aceita uma base e um expoente (com valor padrão 2) e retorna a base elevada ao expoente.

```javascript
// Solução
const potencia = (base, expoente = 2) => Math.pow(base, expoente);

console.log(potencia(3));    // 9
console.log(potencia(3, 3)); // 27
```

Exercício 12: Callback com `setTimeout`

Crie uma função que recebe uma mensagem e um tempo em milissegundos, e usa `setTimeout` com uma arrow function para exibir a mensagem após o tempo especificado.

```javascript
// Solução
function exibirMensagemComAtraso(mensagem, tempoMs) {
  setTimeout(() => {
    console.log(mensagem);
  }, tempoMs);
}

exibirMensagemComAtraso("Esta mensagem aparece após 2 segundos", 2000);
```

Exercício 13: Função de Filtro Personalizada

Implemente uma função `filtrarPor` que recebe um array e uma função de teste, e retorna um novo array com os elementos que passam no teste.

```javascript
// Solução
function filtrarPor(array, funcaoTeste) {
  const resultado = [];

  for (let item of array) {
    if (funcaoTeste(item)) {
      resultado.push(item);
    }
  }

  return resultado;
}

const numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
const pares = filtrarPor(numeros, numero => numero % 2 === 0);

console.log(pares); // [2, 4, 6, 8, 10]
```

Exercício 14: Composição de Funções

Crie uma função `compor` que aceita duas funções e retorna uma nova função que é a composição delas.

```javascript
// Solução
const compor = (f, g) => x => f(g(x));

const dobrar = x => x * 2;
const adicionar5 = x => x + 5;

const dobrarEAdicionar5 = compor(adicionar5, dobrar);
console.log(dobrarEAdicionar5(3)); // 11 (pois (3*2)+5 = 11)
```

Exercício 15: Closure para Contador

Crie uma função que retorna um objeto com métodos para incrementar, decrementar e obter o valor atual de um contador.

```javascript
// Solução
function criarContador(valorInicial = 0) {
  let contador = valorInicial;

  return {
    incrementar: () => ++contador,
    decrementar: () => --contador,
    valor: () => contador
  };
}

const meuContador = criarContador();
console.log(meuContador.incrementar()); // 1
console.log(meuContador.incrementar()); // 2
console.log(meuContador.valor());       // 2
console.log(meuContador.decrementar()); // 1
```

Exercício 16: Função Memoizada

Crie uma função `memoizar` que recebe uma função e retorna uma versão memoizada dela, armazenando resultados para entradas já processadas.

```javascript
// Solução
function memoizar(fn) {
  const cache = {};

  return function(...args) {
    const chave = JSON.stringify(args);

    if (!(chave in cache)) {
      cache[chave] = fn(...args);
    }

    return cache[chave];
  };
}

const calcularFatorial = memoizar(n => {
  if (n <= 1) return 1;
  return n * calcularFatorial(n - 1);
});

console.log(calcularFatorial(5)); // 120
console.log(calcularFatorial(5)); // 120 (usa o valor em cache)
```

Exercício 17: Função Debounce

Implemente uma função `debounce` que limita a frequência com que uma função é executada.

```javascript
// Solução
function debounce(funcao, atraso) {
  let temporizador;

  return function(...args) {
    clearTimeout(temporizador);

    temporizador = setTimeout(() => {
      funcao.apply(this, args);
    }, atraso);
  };
}

const buscar = debounce((termo) => {
  console.log(`Buscando por: ${termo}`);
}, 300);

// Em um cenário real, isso seria chamado em um evento input
buscar("a");
buscar("ap");
buscar("app"); // Apenas esta será executada após 300ms
```

Exercício 18: Função Curry

Implemente uma função `curry` que converte uma função normal em uma função curried.

javascript

```javascript
// Solução
function curry(fn) {
  return function curried(...args) {
    if (args.length >= fn.length) {
      return fn.apply(this, args);
    } else {
      return function(...args2) {
        return curried.apply(this, args.concat(args2));
      };
    }
  };
}

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

const somarCurry = curry(somar);
console.log(somarCurry(1)(2)(3));  // 6
console.log(somarCurry(1, 2)(3));  // 6
console.log(somarCurry(1)(2, 3));  // 6
console.log(somarCurry(1, 2, 3));  // 6
```

Exercício 19: Processador de Eventos

Crie um sistema simples de eventos com funções para registrar, disparar e remover ouvintes de eventos.

javascript

```javascript
// Solução
function criarProcessadorDeEventos() {
  const eventos = {};

  return {
    registrar: (evento, callback) => {
      if (!eventos[evento]) {
        eventos[evento] = [];
      }
      eventos[evento].push(callback);
      return () => this.remover(evento, callback);
    },

    disparar: (evento, dados) => {
      if (!eventos[evento]) return;
      eventos[evento].forEach(callback => callback(dados));
    },

    remover: (evento, callback) => {
      if (!eventos[evento]) return;
      eventos[evento] = eventos[evento].filter(cb => cb !== callback);
    }
  };
}

const processador = criarProcessadorDeEventos();
const onMensagem = mensagem => console.log("Mensagem:", mensagem);
processador.registrar("mensagem", onMensagem);
processador.disparar("mensagem", "Olá mundo!"); // Mensagem: Olá mundo!
```

Exercício 20: Construtor com Prototype

Converta uma função construtora tradicional para usar arrow functions para seus métodos.

```javascript
// Solução
function Pessoa(nome, idade) {
  this.nome = nome;
  this.idade = idade;

  this.apresentar = () => `Olá, meu nome é ${this.nome} e tenho ${this.idade} anos.`;
  this.envelhecer = anos => {
    this.idade += anos;
    return this.idade;
  };
}

const joao = new Pessoa("João", 25);
console.log(joao.apresentar()); // Olá, meu nome é João e tenho 25 anos.
joao.envelhecer(5);
console.log(joao.apresentar()); // Olá, meu nome é João e tenho 30 anos.
```

Nível Avançado

Exercício 21: Implementação de uma Promise

Implemente uma versão simplificada de uma Promise usando funções anônimas.

javascript

```
// Solução
function MinhaPromise(executor) {
  let estado = 'pendente';
  let valor = null;
  let callbacks = {
    then: [],
    catch: []
  };

  function resolver(resultado) {
    if (estado !== 'pendente') return;
    estado = 'realizada';
    valor = resultado;

    callbacks.then.forEach(callback => callback(valor));
    callbacks = { then: [], catch: [] };
  }

  function rejeitar(erro) {
    if (estado !== 'pendente') return;
    estado = 'rejeitada';
    valor = erro;

    callbacks.catch.forEach(callback => callback(valor));
    callbacks = { then: [], catch: [] };
  }

  this.then = function(onRealized) {
    if (estado === 'pendente') {
      callbacks.then.push(onRealized);
    } else if (estado === 'realizada') {
      setTimeout(() => onRealized(valor), 0);
```

```
    }
    return this;
  };

  this.catch = function(onRejected) {
    if (estado === 'pendente') {
      callbacks.catch.push(onRejected);
    } else if (estado === 'rejeitada') {
      setTimeout(() => onRejected(valor), 0);
    }
    return this;
  };

  try {
    executor(resolver, rejeitar);
  } catch (e) {
    rejeitar(e);
  }
}

// Usando nossa implementação
const minhaPromessa = new MinhaPromise((resolve, reject) => {
  setTimeout(() => {
    const sucesso = true;
    if (sucesso) {
      resolve("Operação bem-sucedida!");
    } else {
      reject("Operação falhou!");
    }
  }, 1000);
});
```

```
minhaPromessa
  .then(resultado => console.log(resultado))
  .catch(erro => console.error(erro));
```

Exercício 22: Composição de Componentes

Crie um sistema de composição de componentes onde cada componente é uma função que pode ser combinada com outras.

javascript

```
// Solução
const withBorder = (Component) => (props) => {
  return {
    render: () => {
      const element = Component(props).render();
      return `<div style="border: 1px solid black; padding: 10px">${element}</div>`;
    }
  };
};

const withTitle = (title) => (Component) => (props) => {
  return {
    render: () => {
      const element = Component(props).render();
      return `<h2>${title}</h2>${element}`;
    }
  };
};

const Card = (props) => {
  return {
    render: () => {
      return `<div class="card">${props.content}</div>`;
    }
  };
};

const TitledBorderedCard = withTitle("Meu Título")(withBorder(Card));
const element = TitledBorderedCard({ content: "Conteúdo do cartão" });
console.log(element.render());
```

```
// Output:
// <h2>Meu Título</h2><div style="border: 1px solid black; padding: 10px"><div class="car
```

Exercício 23: Programação Funcional - Pipeline

Implemente uma função `pipeline` que permite encadear operações de transformação de dados.

javascript

```javascript
// Solução
const pipeline = (...funcoes) => {
  return (valorInicial) => {
    return funcoes.reduce((valorAtual, funcao) => {
      return funcao(valorAtual);
    }, valorInicial);
  };
};

// Funções para usar no pipeline
const dobrar = x => x * 2;
const adicionar10 = x => x + 10;
const quadrado = x => x * x;
const converterParaString = x => `O resultado é: ${x}`;

// Criando um pipeline de transformações
const meuPipeline = pipeline(
  dobrar,
  adicionar10,
  quadrado,
  converterParaString
);

console.log(meuPipeline(5)); // "O resultado é: 400" (pois: ((5*2)+10)^2 = 400)
```

Exercício 24: Programação Assíncrona Avançada

Implemente uma função que executa operações assíncronas em série, uma após a outra.

javascript

```javascript
// Solução
const executarEmSerie = async (funcoes, valorInicial) => {
  let resultado = valorInicial;

  for (const funcao of funcoes) {
    resultado = await funcao(resultado);
  }

  return resultado;
};

// Funções assíncronas de exemplo
const buscarUsuario = async (id) => {
  // Simulando uma busca de API
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({ id, nome: `Usuário ${id}` });
    }, 500);
  });
};

const buscarPosts = async (usuario) => {
  // Simulando uma busca de posts
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({
        usuario,
        posts: [`Post 1 do ${usuario.nome}`, `Post 2 do ${usuario.nome}`]
      });
    }, 500);
  });
};
```

```
const formatarResultado = async (dados) => {
  // Simulando processamento
  return new Promise(resolve => {
    setTimeout(() => {
      resolve({
        nome: dados.usuario.nome,
        qtdPosts: dados.posts.length,
        posts: dados.posts
      });
    }, 300);
  });
};

// Executando o pipeline assíncrono
executarEmSerie(
  [
    id => buscarUsuario(id),
    usuario => buscarPosts(usuario),
    dados => formatarResultado(dados)
  ],
  1
)
.then(resultado => console.log(resultado))
.catch(erro => console.error("Erro:", erro));
```

Exercício 25: Gerenciamento de Estado

Implemente um gerenciador de estado simples usando funções e closures.

javascript

```javascript
// Solução
function createStore(reducer, estadoInicial = {}) {
  let estado = estadoInicial;
  const ouvintes = [];

  function getState() {
    return Object.assign({}, estado);
  }

  function dispatch(acao) {
    estado = reducer(estado, acao);
    ouvintes.forEach(ouvinte => ouvinte());
    return acao;
  }

  function subscribe(ouvinte) {
    ouvintes.push(ouvinte);
    return () => {
      const index = ouvintes.indexOf(ouvinte);
      if (index !== -1) ouvintes.splice(index, 1);
    };
  }

  // Despacha uma ação inicial para preencher o estado
  dispatch({ type: '@@INIT' });

  return { getState, dispatch, subscribe };
}

// Exemplo de uso
const contadorReducer = (estado = { valor: 0 }, acao) => {
  switch (acao.type) {
```

```
    case 'INCREMENTAR':
      return { valor: estado.valor + 1 };
    case 'DECREMENTAR':
      return { valor: estado.valor - 1 };
    default:
      return estado;
  }
};

const store = createStore(contadorReducer);

// Inscrever para mudanças
const unsubscribe = store.subscribe(() => {
  console.log("Estado atualizado:", store.getState());
});

// Despachar ações
store.dispatch({ type: 'INCREMENTAR' }); // Estado atualizado: { valor: 1 }
store.dispatch({ type: 'INCREMENTAR' }); // Estado atualizado: { valor: 2 }
store.dispatch({ type: 'DECREMENTAR' }); // Estado atualizado: { valor: 1 }

// Cancelar inscrição
unsubscribe();
```

Exercício 26: Programação Reativa Simples

Implemente um sistema simples de programação reativa usando funções.

javascript

```
// Solução
function Observable() {
  const observers = [];

  return {
    subscribe: (observer) => {
      observers.push(observer);

      return {
        unsubscribe: () => {
          const index = observers.indexOf(observer);
          if (index > -1) {
            observers.splice(index, 1);
          }
        }
      };
    },

    next: (value) => {
      observers.forEach(observer => observer.next(value));
    },

    error: (err) => {
      observers.forEach(observer => {
        if (observer.error) observer.error(err);
      });
    },

    complete: () => {
      observers.forEach(observer => {
        if (observer.complete) observer.complete();
      });
```

```
      observers.length = 0;
    }
  };
}

// Funções para manipular observables
function map(observable, transformFn) {
  const output = Observable();

  observable.subscribe({
    next: (value) => output.next(transformFn(value)),
    error: (err) => output.error(err),
    complete: () => output.complete()
  });

  return output;
}

function filter(observable, predicateFn) {
  const output = Observable();

  observable.subscribe({
    next: (value) => {
      if (predicateFn(value)) {
        output.next(value);
      }
    },
    error: (err) => output.error(err),
    complete: () => output.complete()
  });

  return output;
```

```
}

// Exemplo de uso
const numeros = Observable();

const numerosPares = filter(numeros, n => n % 2 === 0);
const quadrados = map(numerosPares, n => n * n);

const subscription = quadrados.subscribe({
  next: (value) => console.log("Quadrado:", value),
  error: (err) => console.error("Erro:", err),
  complete: () => console.log("Concluído!")
});

// Emitir valores
numeros.next(1);  // Nada é impresso (ímpar)
numeros.next(2);  // "Quadrado: 4"
numeros.next(3);  // Nada é impresso (ímpar)
numeros.next(4);  // "Quadrado: 16"

// Concluir o observable
numeros.complete();  // "Concluído!"
```

Exercício 27: Módulo com Estruturas de Dados Funcionais

Implemente uma lista ligada funcional com operações como map, filter e reduce.

javascript

```javascript
// Solução
const Lista = (() => {
  // Nó vazio como constante singleton
  const EMPTY = Object.freeze({
    isEmpty: true,
    toString: () => "[]"
  });

  // Construtor de nó
  const Cons = (head, tail) => Object.freeze({
    head,
    tail,
    isEmpty: false,
    toString: () => `[${_toString(this)}]`
  });

  // Função auxiliar para toString
  const _toString = (list) => {
    if (list.isEmpty) return "";
    if (list.tail.isEmpty) return `${list.head}`;
    return `${list.head}, ${_toString(list.tail)}`;
  };

  // Criar uma lista a partir de um array
  const fromArray = (array) => {
    if (!array || array.length === 0) return EMPTY;
    return Cons(array[0], fromArray(array.slice(1)));
  };

  // Converter lista para array
  const toArray = (list) => {
    if (list.isEmpty) return [];
```

```
  return [list.head, ...toArray(list.tail)];
};

// Map - aplicar uma função a cada elemento
const map = (list, fn) => {
  if (list.isEmpty) return EMPTY;
  return Cons(fn(list.head), map(list.tail, fn));
};

// Filter - filtrar elementos
const filter = (list, predicate) => {
  if (list.isEmpty) return EMPTY;
  if (predicate(list.head)) {
    return Cons(list.head, filter(list.tail, predicate));
  }
  return filter(list.tail, predicate);
};

// Reduce - agregar elementos
const reduce = (list, fn, initial) => {
  if (list.isEmpty) return initial;
  return reduce(list.tail, fn, fn(initial, list.head));
};

return {
  EMPTY,
  Cons,
  fromArray,
  toArray,
  map,
  filter,
  reduce
```

```
  };
})();

// Exemplo de uso
const numeros = Lista.fromArray([1, 2, 3, 4, 5]);
const dobrados = Lista.map(numeros, x => x * 2);
const filtrados = Lista.filter(dobrados, x => x > 5);
const soma = Lista.reduce(filtrados, (acc, x) => acc + x, 0);

console.log(Lista.toArray(dobrados));   // [2, 4, 6, 8, 10]
console.log(Lista.toArray(filtrados));  // [6, 8, 10]
console.log(soma);  // 24
```

Exercício 28: Biblioteca de Animação

Implemente uma pequena biblioteca para criar animações usando requestAnimationFrame e funções.

javascript

```javascript
// Solução
const Animation = (() => {
  // Armazena todas as animações ativas
  const animations = new Map();
  let nextId = 1;

  // Função para calcular o valor intermediário (easing)
  const linear = (t) => t;
  const easeInOut = (t) => t < 0.5 ? 2 * t * t : -1 + (4 - 2 * t) * t;

  // Função que executa cada quadro da animação
  const tick = (timestamp) => {
    animations.forEach((animation, id) => {
      const { startTime, duration, onUpdate, onComplete, easing, value } = animation;

      // Calcular o progresso (0 a 1)
      const elapsed = timestamp - startTime;
      const progress = Math.min(elapsed / duration, 1);
      const easedProgress = easing(progress);

      // Chamar a função de atualização
      onUpdate(easedProgress, value(easedProgress));

      // Verificar se a animação terminou
      if (progress >= 1) {
        onComplete();
        animations.delete(id);
      }
    });

    // Continuar o loop se houver animações ativas
    if (animations.size > 0) {
```

```
    requestAnimationFrame(tick);
  }
};

// Função para iniciar uma animação
const animate = ({
  duration = 1000,
  from = 0,
  to = 1,
  onUpdate = () => {},
  onComplete = () => {},
  easing = linear
}) => {
  const id = nextId++;
  const startTime = performance.now();

  // Função para calcular o valor atual com base no progresso
  const value = (progress) => from + (to - from) * progress;

  // Armazenar a animação
  animations.set(id, {
    startTime,
    duration,
    onUpdate,
    onComplete,
    easing,
    value
  });

  // Iniciar o loop se for a primeira animação
  if (animations.size === 1) {
    requestAnimationFrame(tick);
```

```
    }

    // Retornar funções para controlar a animação
    return {
      cancel: () => {
        animations.delete(id);
      },
      isActive: () => animations.has(id)
    };
  };

  return {
    animate,
    easing: {
      linear,
      easeInOut
    }
  };
})();

// Exemplo de uso
const elemento = { opacity: 0 };

Animation.animate({
  duration: 2000,
  from: 0,
  to: 1,
  easing: Animation.easing.easeInOut,
  onUpdate: (progress, value) => {
    elemento.opacity = value;
    console.log(`Opacity: ${value.toFixed(2)}`);
  },
```

```
  onComplete: () => {
    console.log("Animação concluída!");
  }
});
```

Exercício 29: Gerenciador de Formulários com Validação

Implemente um sistema para gerenciar formulários com validação usando funções.

javascript

```javascript
// Solução
const FormManager = (() => {
  // Função para criar um campo do formulário
  const createField = (initialValue = '', validators = []) => {
    let value = initialValue;
    let touched = false;
    let dirty = false;

    return {
      getValue: () => value,
      setValue: (newValue) => {
        value = newValue;
        dirty = true;
        return value;
      },
      touch: () => {
        touched = true;
      },
      reset: () => {
        value = initialValue;
        touched = false;
        dirty = false;
      },
      validate: () => {
        return validators.reduce((errors, validator) => {
          const error = validator(value);
          if (error) errors.push(error);
          return errors;
        }, []);
      },
      isTouched: () => touched,
      isDirty: () => dirty
```

```
  };
};

// Função para criar um formulário
const createForm = (fields) => {
  return {
    getField: (name) => fields[name],

    getValue: (name) => fields[name].getValue(),

    setValue: (name, value) => {
      if (fields[name]) {
        fields[name].setValue(value);
      }
    },

    getValues: () => {
      return Object.keys(fields).reduce((values, name) => {
        values[name] = fields[name].getValue();
        return values;
      }, {});
    },

    touch: (name) => {
      if (name) {
        if (fields[name]) fields[name].touch();
      } else {
        Object.values(fields).forEach(field => field.touch());
      }
    },

    reset: () => {
```

```
    Object.values(fields).forEach(field => field.reset());
  },

  validate: () => {
    const fieldErrors = {};
    let hasErrors = false;

    Object.keys(fields).forEach(name => {
      const errors = fields[name].validate();
      if (errors.length > 0) {
        fieldErrors[name] = errors;
        hasErrors = true;
      }
    });
```