Introdução a Orientação a Objetos

Tipos Abstratos de Dados

Enquanto estudamos lógica de programação trabalhamos com os tipos já presentes na linguagem de programação que escolhemos. O que a programação orientada a objetos nos permite fazer é criar novos tipos na linguagem. Chamamos esses tipos de TAD (Tipos Abstratos de Dados).

Esses novos tipos podem ser usados como estruturas de dados, sendo a composição de um ou mais tipos já existentes na linguagem, para representar algo mais complexo, cada tipo usado para armazenar valores é chamado de atributo.

E também podem conter comportamentos, chamados métodos, que nada mais são do que funções que permitem a alteração e leitura dos dados ou interagir com o ambiente externo ao TAD.

Usamos os TADs para mapear de forma computacional as entidades (reais ou abstratas) que fazem parte de um problema que desejamos resolver. Assim, temos uma visão mais realista do problema, utilizamos o mesmo vocabulário do processo real e temos entidades no código que são compatíveis com o que vemos na realidade do problema.

Imagine que precisamos fazer uma agenda, que deve guardar nome, telefone e email de um número arbitrário de pessoas. Com o que conheçemos até o momento, teríamos que fazer 3 vetores, um para o nome, um para o telefone e um para o email.

Quando inserirmos um contato, precisaremos adicionar os dados nos três vetores, e garantir que eles estejam no mesmo índice, pois a posição dos dados deve ser a mesma nos três vetores para podermos agrupá-los como um contato.

Observe que não temos o conceito do contato em lugar nenhum, tampouco o conceito de agenda, tudo está solto no programa e a ligação dos dados é extremamente frágil (basta errar o índice e o nome de um vai aparecer como o telefone de outro).

Em POO (programação orientada a objeto) faríamos diferente, usaríamos entidades que pertencem ao domínio com a nomenclatura utilizada no problema real.

Faríamos então uma entidade (usaremos esse nome por enquanto) chamada Agenda, que será um TAD. Faremos também um TAD para o Contato, que conterá nome, email e telefone.

Agenda se comportará como uma lista de Contatos, contendo métodos que permitam adicionar e remover Contatos, assim como buscar pelo nome.

Assim, centralizamos toda informação do contato em uma única entidade o que acaba com as chances de misturar os dados. E a entidade Agenda tem a responsabilidade de manipular os contatos centralizando as operações que o sistema pode fazer.

O que é importante já perceber é que é muito mais claro falar de uma Agenda que é uma Lista de Contatos, do que falar de três vetores de informação onde os índices seriam os contatos, sem que exista nada chamado contato ou agenda no código.

Essa é a ideia da POO, aproximar o código da "vida real", mapeando processos usando o mesmo vocabulário que é usado na "vida real" e dividindo responsabilidades entre entidades diferentes para não acumular todo o código em um único programa enorme que faz tudo.

Criar TADs não é difícil, e os conceitos que você vai precisar são análogos ao de variáveis e funções, que já vimos na lógica de programação.

No entanto, faremos uma mudança de paradigma de programação, então vamos sair de um mundo linear onde o código roda linha a linha, para um onde os códigos estão dentro das entidades e são elas que executam as funcionalidades do sistema.

Criando Objetos

Normalmente em uma linguagem orientada a objetos, é necessário criar uma classe antes de poder criar objetos. No entanto, como JavaScript é uma linguagem dinâmica e a orientação a objetos é suportada, mas não é o paradigma principal de programação nessa linguagem, é possível criar objetos sem criar classes.

Isso é útil para quando vamos fazer um único objeto, usado em um determinado lugar, mas não precisaremos reaproveitá-lo em outras partes do sistema.

Para criar objetos sem classes, o JavaScript se vale de uma notação de criação de objetos, depois veremos que é quase idêntico ao JSON, que é uma representação textual do objeto.

Essa notação é extremamente simples e concisa. Um objeto é detonado pela abertura de chaves, os atributos são pares chave-valor separados por dois pontos (:) e os métodos são funções.

Por exemplo:

```
const pessoa1 = { nome : "Carlos", idade : 20 };

const quadrado = {
    base : 10,
    altura : 10,
    calcularArea : function() => { return this.base * this.altura; }
};
```

Observe que para nos referirmos a um atributo ou método do objeto de dentro dele mesmo precisamos usar o prefixo *this* para saber que é algo que pertence ao objeto e não alguma variável declarada previamente.

Podemos inclusive criar objetos dentro de objetos ou mesmo vetores sem problemas:

```
const agenda = {
    contatos : [
        {nome : 'contato1', telefone : 'telefone1', email : 'email1@teste.com'},
        {nome : 'contato2', telefone : 'telefone2', email : 'email2@teste.com'},
        {nome : 'contato3', telefone : 'telefone3', email : 'email3@teste.com'},
        {nome : 'contato4', telefone : 'telefone4', email : 'email4@teste.com'}
    ],
    adicionar : function(contato){ this.contatos.push(contato) };
}
```

É simples escrever objetos assim e extremamente comum no javascript, no entanto, para escrever estruturas reutilizáveis é necessário usar classes

Classe e Construtor

Classes são a forma de definir as entidades no nosso sistema. Elas são estruturas capazes de dar origem a infinitos objetos de mesmo tipo.

Para criar uma classe usamos o comando `class` no JavaScript.

```
class <nomedaclasse> {

}
```

O aspecto mais relevante de uma classe é o método construtor, onde indicaremos quais são os atributos que a classe possui e quais são os valores necessários que precisam ser definidos no momento da criação do objeto.

A classe é semelhante a uma "planta baixa" que determina a estrutura e os comportamentos dos objetos criados por ela. Uma classe Pessoa pode indicar que todo objeto tenha um nome, mas não pode indicar qual é esse nome, pois ele será diferente para cada objeto Pessoa criado por ela.

Para criar a classe Pessoa, com nome e idade, as informações necessárias devem ser passadas por parâmetros no construtor e atribuída em atributos precedidos por `this.` :

```
class Pessoa {
    constructor(nome, idade){
        this.nome = nome;
        this.idade = idade;
    }
}
```

Para criar um objeto do tipo Pessoa faremos:

```
const pessoa1 = new Pessoa('Carlos', 20);
console.log(pessoa1);
//Veremos no console:
//Pessoa { nome: 'Carlos', idade: 20 }
```

Sempre que precisar de uma nova Pessoa, basta criar outro objeto:

```
const pessoa2 = new Pessoa('Marta', 26);
console.log(pessoa2);
//Veremos no console:
//Pessoa { nome: 'Marta', idade: 26 }
```

É muito recomendado sempre iniciar nomes de classes com letra maiúscula. Além de manter o código mais organizado, facilita a leitura e entendimento uma vez que cria uma distinção clara entre as classes e variáveis.

Atributos

Restringindo Tipo de Atributos

Vimos na aula anterior que para criar um atributo basta usar o prefixo *this* e dar um nome para o atributo. Isso se dá porque a tipagem dinâmica do javascript também se dá nos atributos. O quer dizer que o tipo do atributo será definido pelo tipo do valor colocado nele.

Quando fazemos estruturas de dados, algumas vezes esse é o comportamento que desejamos, no entanto em outras esse comportamento é inadequado. Existem estruturas onde precisamos restringir os tipos dos atributos para que a estrutura funcione corretamente.

Por exemplo:

```
class Quadrado{
    constructor(base, altura){
        this.base = base;
        this.altura = altura;
    }
}
```

Quando formos adicionar um método para calcular área nessa classe, precisamos garantir que os valores de base e altura sejam números ou então o cálculo será impossível, lembre-se que nada impede que sejam passadas strings por exemplo! Veja:

```
const quadrado = new Quadrado(3, 4);
console.log(quadrado);

const quadrado2 = new Quadrado('teste', 'teste2');
console.log(quadrado2);
```

Ambos funcionam! Para isso, é necessário fazer uma lógica condicional, podemos usar isNaN que é uma função que verifica se o valor não é número:

```
class Quadrado{
    constructor(base, altura){
        if(isNaN(base) || isNaN(altura)) throw "Base e altura precisam ser números!";
        this.base = base;
        this.altura = altura;
    }
}
```

Assim, nossa classe passa a produzir um erro se mal utilizada.

Atributos Opcionais

Podemos ter atributos opcionais em uma classe também, não é necessário, mas aconselhável passar todos os valores necessários no construtor.

Por exemplo, imagine que o quadrado possa ter uma cor, que é opcional.

Não colocaremos a cor nos parâmetros do construtor, mas criaremos um atributo para ela com o valor "undefined" que significa não definido. A cor poderá ser alterada depois de fora da classe.

```
class Quadrado{
    constructor(base, altura){
        if(isNaN(base) || isNaN(altura)) throw "Base e altura precisam ser números!";
        this.base = base;
        this.altura = altura;
        this.cor = undefined;
    }
}

const quadrado = new Quadrado(3, 4);
console.log(quadrado);
//Quadrado { base: 3, altura: 4, cor: undefined }
quadrado.cor = 'azul';
console.log(quadrado);
//Quadrado { base: 3, altura: 4, cor: 'azul' }
```

Observe que diferentemente da base e altura, que podemos validar antes, no caso da cor o usuário pode passar o valor que ele quiser e não temos como restringir porque não estamos no contexto de método como acontece no construtor.

Mais para frente veremos como fazer essa restrição usando um conceito da POO chamado encapsulamento e métodos de acesso (que no JavaScript chamam Accessors). Mas antes, vamos ver como implementar métodos em classes.

Métodos

Métodos dão aos nossos objetos o poder de executar códigos, o que chamamos de comportamento há algumas aulas. Tudo o que você aprendeu em lógica de programação até o momento pode ser utilizado dentro dos métodos (e somente dentro dos métodos) quando estamos no contexto orientado a objetos.

Métodos são computacionalmente idênticos às funções. Chamamos de métodos e não funções por dois motivos:

1. Por estar no contexto POO e no interior de uma classe.
2. Por ele poder manipular o estado interno de um objeto.

Leia "estado" acima como o conjunto de valores dos atributos.

Para declarar um método em uma classe basta fazer, após o construtor, uma função sem usar a palavra `function` usamos apenas o nome. Também não podemos usar arrow functions nesse contexto por conta do mesmo problema com `this.` que tivemos nos objetos.

```
class Quadrado{
    constructor(base, altura){
        if(isNaN(base) || isNaN(altura)) throw "Base e altura precisam ser números";
        this.base = base;
        this.altura = altura;
        this.cor = undefined;
    }
    calcularArea() {
        return this.base * this.altura;
    }
}

const quadrado = new Quadrado(3, 4);
console.log(quadrado);
//Quadrado { base: 3, altura: 4, cor: undefined }
console.log(quadrado.calcularArea());
//12
```

Métodos podem ler ou alterar o estado interno do objeto, assim como podem fazer qualquer computação com os valores dos atributos e até mesmo chamar outros métodos do mesmo objeto.

Alguns métodos podem até criar outros objetos ou funções, dependendo da necessidade. Alguns padrões como Factory dependem dessa capacidade. Imagine por exemplo uma classe de configuração onde você coloca vários dados e no final ela te devolve uma função que abre uma conexão com um banco de dados usando os dados fornecidos.

No entanto, se métodos estiverem fazendo computações independentes de qualquer informação pertencente ao objeto, existe grandes chances de que eles estejam no lugar errado. Nesse caso, eles poderíam estar em outro objeto ou no próprio script como uma função.

Vejamos alguns exemplos do que métodos podem fazer com a classe Quadrado:

```
class Quadrado{
  constructor(base, altura){
    if(isNaN(base) || isNaN(altura)) throw "Base e altura precisam ser números";
```

```
    this.base = base;
    this.altura = altura;
    this.cor = undefined;
  }
  calcularArea() {
    return this.base * this.altura;
  }
  duplicarParaDireita(){
    this.base = this.base * 2;
  }
  duplicarParaBaixo(){
    this.altura = this.altura * 2;
  }
  imprimir(){
    return `<div style='width:${this.base}px;height:${this.altura}px;background-
color:${this.cor || "blue"}'></div>`;
  }
}

const quadrado = new Quadrado(3, 4);
console.log(quadrado);
//Quadrado { base: 3, altura: 4, cor: undefined }
console.log(quadrado.calcularArea());
//12

quadrado.duplicarParaBaixo();
console.log(quadrado);
//Quadrado { base: 3, altura: 8, cor: undefined }

quadrado.duplicarParaDireita();
console.log(quadrado);
//Quadrado { base: 6, altura: 8, cor: undefined }

console.log(quadrado.imprimir());
//<div style='width:6px;height:8px;background-color:blue'></div>
```

Encapsulamento

Encapsulamento é o conceito de negar o acesso aos atributos de uma classe diretamente, seja para leitura ou escrita. A maioria das linguagens orientadas a objetos utilizam algum comando como `private` por exemplo para restringir esse acesso.

O JavaScript não é uma linguagem orientada a objetos de nascença, o suporte a conceitos mais avançados em POO só foi adicionado recentemente. JavaScript até hoje ainda trata classes como funções, mesmo tendo a palavra `class` e uma sintaxe especial para montar classes.

A parte boa é que podemos utilizar o conhecimento em funções para realizar coisas nas classes que suprem a ausência de alguns conceitos. Por exemplo, variáveis dentro de funções não são acessíveis de fora da função, então poderíamos fazer encapsulamento usando elas.

Veja esse exemplo que usa uma função em vez de `class`:

```
function Quadrado(base, altura){
  this.base = base;
  this.altura = altura;
  let cor = 'blue';
}

const quadrado = new Quadrado(3,4);

console.log(quadrado);
//Quadrado { base: 3, altura: 4 }
```

```
console.log(quadrado.cor);
//undefined
quadrado.cor = 'red';
//não altera cor dentro do quadrado
```

Aqui usamos `let` no lugar de `this.` para indicar que a cor é uma informação privada da classe Quadrado, de forma que ela não está acessível para leitura ou alteração de fora dessa classe.

Também podemos usar um factory function, uma função que cria um objeto, para encapsular informações seria assim:

```
function createQuadrado(base, altura){

  let cor = 'blue';

  return {
    base,
    altura,
    getCor : function() { return cor; }
  };

}

const quadrado = createQuadrado(3,4);
console.log(quadrado);
//{ base: 3, altura: 4, getCor: [Function: getCor] }
console.log(quadrado.cor);
//undefined
quadrado.cor = 'red';
console.log(quadrado.cor);
//red
console.log(quadrado.getCor());
//blue
```

Criamos uma função que constrói e retorna um objeto. Essa função tem uma variável interna chamada cor, que contém o valor 'blue' e passamos por parâmetro o valor da base e da altura.

Em seguida retornamos um objeto contendo a base e a altura, mas não a cor. Fizemos um método que lê o valor privado da cor e o retorna.

Observe que quando tentamos usar `quadrado.cor` ele diz *undefined*. Pois não há cor no quadrado.

Quando usamos `quadrado.cor = 'red'` isso não alterou a cor do quadrado, o que aconteceu é que ele inseriu um novo campo chamado cor no quadrado (que não tinha esse campo) com o valor 'red'. Observe que nossa função que lê a cor encapsulada ainda retorna 'blue'.

Muito cuidado com isso! Linguagens dinâmicas como JavaScript ou Python, permitem a inserção de novos atributos ou métodos em objetos pré-existentes. Isso torna erros de digitação particularmente problemáticos, porque em vez de trocar o valor de um atributo existente, criamos um novo com o nome incorreto.

Quanto a sintaxe de classe, estima-se que terá em um futuro próximo uma sintaxe usando # na frente de variáveis privadas. Hoje isso é suportado em alguns navegadores maiores, mas ainda não está oficializado.

O que podemos fazer é no método construtor declarar uma variável como fizemos na factory function acima:

```
class Quadrado{
```

```
  constructor(lado, altura){
    let cor = 'blue';
    this.lado = lado;
    this.altura = altura;
    this.getCor = () => { return cor; };
  }
}

const quadrado = new Quadrado(3,4);
console.log(quadrado);
//Quadrado { lado: 3, altura: 4, getCor: [Function], setCor: [Function] }
console.log(quadrado.getCor());
//blue
quadrado.cor = 'red'; //errado! cria um atributo cor a mais no objeto!
console.log(quadrado.cor);
//red
console.log(quadrado.getCor());
//blue
```

O problema dessa técnica é que o escopo de 'cor' na classe é apenas o do construtor, isso nos obriga a declara todos os métodos que precisam de acesso à 'cor' dentro do construtor. Não podemos nesse caso usar a sintaxe comum de métodos que usamos na aula anterior.

Quando o encapsulamento complica mais do que ajuda, podemos recorrer a uma outra técnica, que é padronizar o acesso aos valores dos atributos de forma que voltamos a ter o controle sobre atribuições e leituras, como fazíamos com os valores vindos dos parâmetros do construtor.

Para isso usaremos um conceito chamado em algumas linguagens de métodos de acesso, ou propriedades, ou como chamamos no JavaSript "Acccessors".

Métodos de Acesso

Métodos de Acesso em linguagens como o Java são métodos que permitem a leitura e/ou escrita (a critério do programador) de atributos privados. Podemos criar métodos de acesso ao molde do que fizemos na aula passada:

```
class Quadrado{

  constructor(lado, altura){
    let cor = 'blue';
    this.lado = lado;
    this.altura = altura;
    this.getCor = () => { return cor; };
    this.setCor = (c) => cor = c;
  }
}
```

Nesse exemplo, getCor retorna o valor da cor enquanto o setCor altera o valor da cor. No entanto, não podemos impedir que alguém por engano crie um atributo cor no objeto e passe um valor para ele, como vimos anteriormente.

Podemos usar o conceito de propriedades para nos auxiliar nesse caso. De forma simplificada, propriedades são a soma do conceito de atributos e métodos de acesso. Uma propriedade seria, portanto, um atributo que tem métodos de acesso.

Como uma propriedade não é uma variável da função construtora, como fizemos acima, ela pertence ao objeto (o que impede que ela seja criada por engano) e tem seu acesso restrito por métodos, que no javascript são chamados de Accessors:

```
class Quadrado{

  constructor(lado, altura){
    this._cor = 'blue';
    this.lado = lado;
    this.altura = altura;
  }

  get cor() { return this._cor; }
  set cor(cor) { this._cor = cor; }
}

const quadrado = new Quadrado(3,4);
console.log(quadrado);
//Quadrado { cor: 'blue', lado: 3, altura: 4 }
console.log(quadrado.cor);
//blue
quadrado.cor = 'red';
console.log(quadrado.cor);
//red
```

A parte chata de usar os Accessors é que eles que devem ter o nome da propriedade e esse nome não pode colidir com o nome de um atributo. Isso nos obrigou a renomear a cor como atributo para "_cor", o underscore sempre foi usado por programadores das antigas para simbolizar private.

Os Accessors permitiram a leitura e a atribuição de valores na propriedade cor, mas você pode estar pensando, mas qual a diferença disso e de não fazer o get/set?

Nesse exemplo, só há uma diferença. A atribuição e a leitura da propriedade passa por métodos em vez de ser feita como uma variável. Lembre-se do que já falamos, em POO métodos permitem a aplicação de tudo o que já fizemos em lógica de programação. Isso quer dizer que podemos modificar a lógica do get e do set para fazer praticamente qualquer coisa.

Vamos explorar isso usando aquele exemplo de algumas aulas atrás:

```
class Quadrado{
    constructor(base, altura){
        if(isNaN(base) || isNaN(altura)) throw "Base e altura precisam ser números";
        this.base = base;
        this.altura = altura;
        this.cor = undefined;
    }
}
```

Temos uma classe quadrado que recebia base e altura por parâmetros e validava se ambos eram valores numéricos antes de atribuir nos atributos.

No entanto, para a cor, que não era passada nos parâmetros do construtor, não conseguimos fazer nenhuma validação se o valor passado é válido. Porém usando propriedades e accessors podemos fazer isso!

```
class Quadrado{
  constructor(base, altura){
    if(isNaN(base) || isNaN(altura)) throw 'Base e altura precisam ser números';
    this.base = base;
    this.altura = altura;
```

```
    this._cor = undefined;
  }
  get cor() { return this._cor; }
  set cor(cor) {
    if(cor !== 'red' && cor !== 'green' && cor !== 'blue'){
      throw 'O valor da propriedade cor deve ser "red", "green" ou "blue"';
    }
    this._cor = cor;
  }
}

const quadrado = new Quadrado(3,4);
console.log(quadrado);
//Quadrado { base: 3, altura: 4, _cor: undefined }
console.log(quadrado.cor);
//blue
quadrado.cor = 'red';
console.log(quadrado.cor);
//red
quadrado.cor = 'white';
//O valor da propriedade cor deve ser "red", "green" ou "blue"
```

Aqui limitamos as atribuições à propriedade cor para os valores "green", "blue" ou "red" e usamos `!==` com dois iguais para garantir que passados como strings.

Conseguimos o que queríamos, mas pouca coisa no JavaScript é perfeita. Linguagens que adicionaram suporte a POO, mas que não tem sua arquitetura projetada para POO desde o princípio tem dificuldades com encapsulamento, isso acontece no JavaScript e no Python.

No exemplo acima, ainda podemos burlar o accessor usando quadrado._cor = 'white'. Sendo assim, a sintaxe de classe do javascript não tem uma forma segura de garantir o encapsulamento.

Para ter uma garantia maior teremos que usar uma factory function combinada com get e set.

```
function createQuadrado(base, altura){
  if(isNaN(base) || isNaN(altura)) throw 'Base e altura precisam ser números';
  let _cor = undefined;
  return {
    base,
    altura,
    get cor() {return _cor; },
    set cor(cor) {
      if(cor !== 'red' && cor !== 'green' && cor !== 'blue'){
        throw 'O valor da propriedade cor deve ser "red", "green" ou "blue"';
      }
      _cor = cor;
    }
  };
}

const quadrado = createQuadrado(3,4);
console.log(quadrado);
//{ base: 3, altura: 4, cor: [Getter/Setter] }
console.log(quadrado.cor);
//blue
quadrado.cor = 'red';
console.log(quadrado.cor);
//red
quadrado.cor = 'white';
//O valor da propriedade cor deve ser "red", "green" ou "blue"
```

Em suma, a orientação a objetos no JavaScript não é perfeita. Se você estiver vindo de uma linguagem nativamente orientada a objetos como C++, C# ou Java isso tudo pode parecer muito estranho.

Nossa recomendação é não se apoiar demais nesses conceitos de encapsulamento. Geralmente usamos esses conceitos apoiados em convenções dentro do time de programadores para garantir um pouco mais de adequação aos conceitos.

Herança e Polimorfismo

Herança

JavaScript tem o conceito de herança como a maioria das linguagens de programação orientadas a objetos. Que é criar uma nova classe que herda os atributoes e métodos de uma classe pai. A nova classe, por sua vez, é chamada de classe filha.

A sintaxe para fazermos herança em JavaScript é a mesma utilizada em Java, ou seja, a palavra `extends` denota a herança e a palavra `super` refere-se a super classe que foi herdada.

```
class Pessoa{
    constructor(nome, idade){
        this.nome = nome;
        this.idade = idade;
    }
}

class Cidadao extends Pessoa{
    constructor(nome, idade, cpf, rg){
        super(nome, idade);
        this.cpf = cpf;
        this.rg = rg;
    }
}
```

No exemplo, a classe `Cidadao` herda a classe `Pessoa`. Usamos `super` como método para invocar o construtor da super classe `Pessoa` e deixar que ele atribua o `nome` e a `idade`.

Podemos, por meio do `super.` acessar qualquer atributo ou método da super classe.

Polimorfismo

Polimorfismo é uma conceito vindo do grego poli morfos, que significa "múltiplas formas".

Ele se aplica na orientação a objetos no uso de Herança e na implementação de interfaces (que não é possível em JavaScript).

Quando fazemos herança, nossa classe passa a ter vários tipos, assim como os objetos criados por ela.

Os tipos são: O tipo da própria classe, denotado pelo mesmo nome da classe e o tipo de qualquer super classe que ela herde.

No exemplo, `Cidadao` é `Cidadao` e `Pessoa`.

Veja que usamos o verbo ser aqui "`Cidadao` **É** `Cidadao` e **É** `Pessoa`.

Essa é uma boa regra para validar herança inclusive, quando você não puder dizer que a sub classe **É** a super classe, ela não será candidata para herança.

Em linguagens dinâmicas, o polimorfismo de classes e objetos não é tão imprescindível e necessário como acontece em linguagens estáticas. No entanto, você sempre poderá verificar se um objeto tem um determinado tipo usando `instanceof`.

```
const cidadao = new Cidadao('teste', 20, '00000', '11111');
console.log(
  cidadao instanceof Cidadao, //true
  cidadao instanceof Pessoa   //true
);
```

Essa técnica é muito usada quando um objeto de determinado tipo deve ser passado por parâmetro para uma função ou método por exemplo.

Usando `instanceof` podemos validar se o que nos foi passado é de fato um objeto da classe que esperamos. Em linguagens dinâmicas, quando criamos bibliotecas para outras pessoas usarem, esse tipo de validação é praticamente obrigatório para não termos problemas com tipos indevidos.

Fechamento Lógica de Programação

Nosso próximo tópico será a biblioteca React, nela utilizaremos os conceitos vistos em orientação a objetos em componentes feitos a partir de classes.

Os conceitos de lógica de programação serão usados para gerenciar a "reação" do software às interações do usuário e durante o ciclo de vida do componente.

Aproveite esse momento para se familiarizar com a documentação do javascript, o melhor recurso para isso é o site MDN:

https://developer.mozilla.org/

Lá você encontrará a documentação completa da linguagem e os conceitos inseridos na atualização de 2015 chamada ES6, que são vastamente utilizados.

Outra referência útil é: **https://www.w3schools.com/jsref/**

Busque sempre exercitar o máximo possível!