**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.

**Obs:**Os cursos digitais dos processos seletivos não contêm os exercícios construtor.

# 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' }

# 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>