**Orientação à Objetos**

**Tipos Abstratos de Dados (TAD)**

TAD são formas de representar novos tipos de dados através da Programação Orientada a Objetos. Esses tipos abstratos de dados (TAD) podem servir de estruturas de dados. São também formados como um conjunto de dados por um ou mais tipos de dados já existentes (strings, inteiros, booleanos). Desta forma é possível criar algo mais complexo. Cada tipo de dado inserido é chamado de atributo. Podem também serem formados por funções que são chamadas de métodos.

Através dos TADs podemos mapear de forma computacional as entidades (reais ou abstratas) que fazem parte de um problema que se deseja resolver. Desta forma conseguimos ter uma visão mais concreta do problema, onde ao ser utilizado o mesmo vocabulário do processo real existem assim no código entidades compatíveis com a realidade do problema.

Por exemplo necessitamos criar um código que armazena contatos onde serão guardados nome, telefone e e-mail. Desta forma criamos uma entidade chamada “Agenda” (TAD) que irá armazenar uma lista com outras entidades chamadas “Contato” (TAD). Dentro de Agenda teremos métodos que permitam adicionar, remover, buscar Contatos, entre outros. Desta forma centralizamos todas as informações de Contatos em uma única entidade, impedindo assim que as informações se misturem ou se percam. Na entidade “Agenda” centralizamos então todas as operações que o sistema pode fazer. É mais fácil falar de uma entidade chamada “Agenda” que possua uma lista de “Contatos” do que falar de toda essa informação espalhada por vários vetores diferentes.

A principal função da Programação Orientada a Objetos é exatamente isso, torar o código o mais próximo da vida real possível, estruturando processos com os mesmos nomes utilizados na vida real e dividindo a responsabilidade entre diferentes entidades para não centralizar todo o código em um único programa enorme que faz tudo.

**Criando Objetos**

Normalmente é necessário criar uma classe antes de criar objetos. Porém por Javascript ser dinâmica, suportar POO e por não ser o paradigma principal da linguagem é possível a criação de objeto sem criar classes. Isso facilita na criação de objetos únicos que são usados apenas em lugares determinados no código, não os reaproveitando em outras partes do sistema. Exemplos de criação de objetos sem criação de classes:

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

**const** quadrado = {

    base : 10,

    altura : 10,

    calcularArea : **function**() **=>** { return this.base \* this.altura; }

};

Para se referir a um atributo ou método dentro do objeto é necessário a utilização do this para que ser entendido que é algo que pertence ao objeto e não uma variável declarada fora dele. Pode-se criar objetos de dentro de outro objeto também.

Para que o objeto seja reutilizável, ou seja, capaz de ser utilizado em outros pontos do código, se faz a necessário a utilização de classes.

Exemplos de atributos (propriedades) : carro -> cor, marca, modelo, ano de fabricação, etc.

Exemplos de métodos (coisas que o objeto faz) : carro -> ligar, desligar, ligar rádio, correr, parar, etc.

**Classe e Construtor**

Classes são a forma de definir entidades no sistema. São estruturas que dão origem a infinitos objetos de mesmo tipo.

**class** <nomedaclasse> {

}

Seu aspecto mais relevante é o método construtor, onde são indicados quais atributos que a classe possui e que valores são necessários e que precisam ser definidos no momento da criação do objeto. Ela é uma espécie de “planta baixa” onde é determinada a estrutura que o objeto precisa ter para ser criado além dos seus comportamentos. Por exemplo uma classe pessoa pode indicar que todo objeto deve possuir nome, porém não pode indicar qual nome deve ser, porque cada objeto Pessoa possui um nome diferente para si.

**class** Pessoa {

**constructor**(nome, idade){

        this.nome = nome;

        this.idade = idade;

    }

};

Para criar o objeto do tipo pessoa:

**const** pessoa1 = new Pessoa('Carlos', 20);

console.log(pessoa1);

*//Veremos no console:*

*//Pessoa { nome: 'Carlos', idade: 20 }*

**const** pessoa2 = new Pessoa('Marta', 26);

console.log(pessoa2);

*//Veremos no console:*

*//Pessoa { nome: 'Marta', idade: 26 }*

Inicia-se o nome de classes por boas práticas com a primeira letra maiúscula.

Basicamente CLASSE é a ideia de um objeto, sendo assim ela é a construtora do objeto. Sendo assim só precisamos passar para ela os atributos que determinado objeto possui.

**Atributos**

**Restringindo Tipo de Atributos**

É necessário verificar se o tipo de dado recebido é o tipo correto, por exemplo altura deve ser um tipo numérico e não uma string. Para isso podemos utilizar a função isNaN. Assim a classe informa se for um tipo de dado incorreto.

**class** Quadrado{

**constructor**(base, altura){

        if(isNaN(base) || isNaN(altura)) throw "Base e altura precisam ser números!";

        this.base = base;

        this.altura = altura;

    }

};

**Atributos Opcionais**

Há a possibilidade de ter atributos que sejam opcionais na classe, não é necessário, mas é aconselhável que todos os atributos sejam passados ao construtor no momento da criação do objeto.

Por exemplo um leve em consideração que o quadrado possua uma cor, isso é um atributo opcional. Não será passado como parâmetro para o construtor a cor do quadrado, mas o atributo será criado com o undefined, sendo possível sua alteração 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.

**Métodos**

É o que possibilita aos objetos executarem códigos. Tudo o que foi passado em lógica de programação pode ser implementado aos métodos. No contexto de POO essas implementações só podem ocorrer dentro dos métodos. Computacionalmente idênticos às funções.

São chamados de métodos e não funções por estarem no contexto de POO e por poder manipular o estado interno de um objeto. Entenda como “estado” como o conjunto de valores dos atributos.

Nesse contexto arrow functions não podem ser utilizadas.

**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*

Possuem o poder de alterar o estado interno do objeto, além de poderem computar qualquer valor dos atributos e chamar outros métodos pertencentes ao mesmo objeto.

**Encapsulamento**

Uma coisa encapsulada é uma coisa que está fechada, onde eu que estou fora não consigo ter acesso ao que está dentro. Encapsulamento é o conceito de negar o acesso aos atributos de uma classe diretamente, tanto para leitura quanto para escrita.

O JavaScript não é uma linguagem OO 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, tais como o PRIVATE, utilizado na maioria das linguagens OO. Por exemplo, variáveis dentro de funções não são acessíveis de fora da função, então poderíamos fazer encapsulamento usando-as. Por exemplo o uso de FUNCTION no lugar 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*

Usando o let no lugar de this fazemos com que a variável fique restrita apenas a classe Quadrado, tornando-a inacessível tanto para leitura quanto para escrita fora da classe.

Também é possível utilizar um factory function, uma função que cria um objeto, para encapsular informações.

**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*

Aqui há 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. Para evitar isso basta inserir no método constructor declarar uma variável com o nome cor.

**class** Quadrado{

**constructor**(lado, altura){

**let** cor = 'blue';

      this.lado = lado;

      this.altura = altura;

      this.getCor = () **=>** { return cor; };

    }

  }

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.

Quando o encapsulamento se torna complicado assim pode-se recorrer ao conceito de métodos de acessou, ou propriedades, chamado no JS de “Accessors”.

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.

**Métodos de Acesso**

São métodos que permitem a leitura e/ou escrita em atributos privados.

class Quadrado{

  constructor(*lado*, *altura*){

    let cor = 'blue';

*this*.lado = *lado*;

*this*.altura = *altura*;

*this*.getCor = () => { return cor; };

*this*.setCor = (*c*) => cor = *c*;

  }

}

No exemplo acima getCor permite a leitura do atributo privado cor. Por sua vez o método setCor permite a escrita (alteração) do atributo privado cor. Com esse método, porém, não há como impedir que o usuário possa criar, por engano, outro atributo chamado cor para o objeto. Desta forma utilizamos o conceito de propriedades para que possamos solucionar esse caso. De forma simples PROPRIEDADES são a soma dos atributos do objeto e seus métodos de acesso. Portanto uma PROPRIEDADE é um atributo que possuí métodos de acesso.

Como uma PROPRIEDADE não é uma variável da função construtora, ela pertence ao objeto, impedindo assim que seja criada por engano. Seu acesso é restrito por métodos, que em JS denominam-se Accessors.

class Quadrado{

  constructor(*lado*, *altura*){

*this*.\_cor = 'blue';

*this*.lado = *lado*;

*this*.altura = *altura*;

  }

  get cor() { return *this*.\_cor; } //ACCESSOR de leitura

  set cor(*cor*) { *this*.\_cor = *cor*; }//ACCESSOR de escrita

}

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

Pelo fato de Accessors terem a obrigatoriedade de levarem o nome da propriedade a qual estão alterando dentro do objeto, sendo desta forma ilegíveis ao usuário se são privados ou não, utilizamos do UNDERSCORE (\_cor) para alterar o atributo do objeto, isso era usado antigamente por programadores para simbolizar o private do atributo.

Através dos ACCESSORs (get e set) conseguimos fazer a leitura e atribuição de valores nesses atributos que são privados. Sendo assim a atribuição e leitura da propriedade passam a se tornar métodos ao invés de serem realizadas como variável. Pelo fato de em POO métodos possibilitarem a implementação de qualquer lógica de programação, podemos modificar as lógicas de get e set para praticamente qualquer coisa que quisermos. Por exemplo:

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"

No exemplo acima temos uma classe Quadrado, que antes de atribuir os valores recebidos aos seus atributos, verifica se são do tipo de dado correto. Porém para o atributo cor, que não era passado como parâmetro ao constructor, não conseguimos fazer uma validação. Através das PROPRIEDADES e ACCESSORs isso é possível.

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"';

     } else {

*this*.\_cor = cor;

   }

}

Aqui limitamos as atribuições à propriedade cor para os valores "green", "blue" ou "red" e usamos !== com dois iguais para garantir que sejam passados como strings. Porém em linguagens cujo paradigma inicial de OO e que adicionaram suporte a OO posteriormente possuem dificuldades com encapsulamento, esse é o caso de JS e Python. No código acima ainda é possível burlar o accessor utilizando quadrado.\_cor = ‘white’. Por esse motivo a sintaxe de classes em JS não possuí uma forma segura de garantir encapsulamento.

Para uma maior garantia de segurança no encapsulamento é necessário utilizar factory functions combinadas ao 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"

Mas o que são FACTORY FUNCTIONS? É qualquer função que não é uma classe, não é um construtor, não é uma função construtora, mas retorna um novo objeto. Em JS qualquer função pode retornar um objeto, e quando a função faz isso sem a obrigatoriedade da palavra new após a declaração de uma variável ou constante, ela é uma factory function.

Com factory functions é possível criar e compor objetos na aplicação web com código mais simples em comparação com um design de código que exige a utilização de classes, construtors e o operador new.

Em resumo OO em JS não é perfeita. Recomenda-se então não se apoiar tanto nesses conceitos de encapsulamento. Esses conceitos são mais utilizados como cultura em times de programadores para garantir que estejam mais adequados aos conceitos.

**Herança e Polimorfismo**

Como a maioria da linguagens de programação possuem o conceito de HERANÇA atrelados a elas, em JS não seria diferente. Seu conceito é que classes filhas herdem de uma classe pai atributos e métodos, sendo esta, pela herança, então chamada de classe filha.

class Pessoa{ //classe pai

  constructor(*nome*, *idade*){

*this*.nome = *nome*;

*this*.idade = *idade*;

  }

}

class Cidadao extends Pessoa{ //classe filha

  constructor(*nome*, *idade*, *cpf*, *rg*){

*super*(*nome*, *idade*);

*this*.cpf = *cpf*;

*this*.rg = *rg*;

  }

}

*Extends* denota a herança e a palavra *super*, por sua vez refere-se a superclasse que foi herdada e seus atributos e métodos. Através do *super* invocamos o construtor da superclasse PESSOA e através dele atribui-se o nome e idade a classe CIDADAO. Por ele também acessamos qualquer atributo ou método da superclasse.

Já o conceito de Polimorfismo se aplica em OO no uso de Herança e na implementação de interfaces, sendo esse último não suportado em JS. Quando herança é utilizada no código, nossa classe passa a ter vários tipos atrelados a ela, assim como objetos criados por ela.

Seus tipos são: o tipo da própria classe e o tipo de qualquer superclasse que ela herde. Por exemplo acima CIDADAO é do tipo CIDADAO e do tipo 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 indispensável e necessário como em linguagens estáticas. Porém, é possível verificar o tipo do objeto utilizando o *instanceof.*

const cidadao = **new** Cidadao('teste', 20, '00000', '11111');

console.log(

  cidadao instanceof *Cidadao*, //true

  cidadao instanceof *Pessoa*   //true

);

Por ele é possível fazer a validação se o que foi passado é realmente o tipo de objeto que estamos aguardando. Em linguagens dinâmicas, quando criamos bibliotecas para outras pessoas usarem, esse tipo de validação é praticamente obrigatório para não termos problemas com tipos indevidos.

Uma habilidade de extrema importância para o programador é aprender a ler a documentação da linguagem que está sendo utilizada. No site abaixo é possível encontrar diversas documentações de linguagens de programação.

developer.mozilla.org/pt-br

**Bibliografia**

<https://www.youtube.com/watch?v=4APvzc9qdN8>

<https://www.youtube.com/watch?v=wQfZ4cyNyqg>