Introdução

01

+ {OOP}

Instalar todas as dependências e entender os conceitos de OO

O que é Orientação a Objetos?

- Um **paradigma** de programação baseado em objetos;
- Onde os **objetos** interagem entre si;
- E o **objeto** é uma instância de uma **classe**;
- Auxilia no reaproveitamento de código;
- Torna o código menos confuso em relação ao procedural;
- D**esign Patterns**;

+ {OOP}

OO x mundo real

- Uma classe Carro representa todos os carros do mundo;
- Porém cada Carro distingue do outro, sendo assim eles são objetos;
- Tendo características (propriedades) e funcionalidades próprias (métodos);
- O Carro pode ter 4 ou 2 portas (propriedades);
- Podemos acelerar ou frear com o Carro (métodos);

+ {OOP}

OO e JavaScript

- JavaScript não é uma linguagem baseada em classes
- Porém podemos utilizar a OOP nela
- Na versão ES2015 foi inserida a funcionalidade de Classe no JS
- JS possui Prototypes, isso faz com que todos os objetos tenham um pai;
- Além dos seus objetos built-in como: Number, Boolean, Array, Object, Error e etc...

+ {OOP}

Fazendo download de um editor

- https://code.visualstudio.com/

Como executar o JS em um navegador

- Criar o arquivo index.html e adicionar JavaScript;

+ {OOP}

Executando JS direto do console

- Digitar código JavaScript no console do Chrome

Outra alternativa para rodar JS

- https://jsfiddle.net/

Como eu aconselho você a seguir o curso

- Use a abordagem de: editor de texto + navegador;
- Salve todas as aulas em arquivos separados para consultar posteriormente;
- Cada aula crie um exemplo a mais com as suas ideias para praticar;

+ {OOP}

Introdução

01

Conclusão da unidade

+ {OOP}

Objetos

02

+ {OOP}

Aprender a criar objetos, propriedades, metodos e assuntos relacionados

O que é um objeto?

- Um tipo de dado que possui duas características;
- **Propriedades:** características do objeto;
- **Métodos:** ações do objeto;
- O protagonista na programação orientada a objetos;

+ {OOP}

Como criar um objeto

- No JavaScript podemos criar um objeto abrindo e fechando chaves;
- O objeto parece com um array de chave e valor;
- Podemos atribuir o objeto a uma variável;

```
let carro = {
  portas: 4
}
```

+ {OOP}

Propriedades

- As propriedades são as características dos objetos;
- Por ex: cor, portas, nome, marca e etc;
- Podemos iniciar um objeto com várias propriedades;
- E acessá-las para resgatar os seus valores;

```
let carro = {
  portas: 4,
  cor: 'verde'
}

console.log(carro.portas);
console.log(carro['cor']);
```

+ {OOP}

Tipos de dados e propriedades

- As propriedades aceitam qualquer tipo de dados do JavaScript
- Booleanos, numbers, strings e arrays

```
let carro = {
  portas: 4,
  cor: "verde",
  opicionais: [
    "teto solar", "blindagem", "ar condicionado"
  ],
  revisado: true
}

console.log(carro.portas);
console.log(carro["cor"]);
console.log(carro.revisado);
console.log(carro.opicionais);
```

+ {OOP}

Propriedades com mais de uma palavra

- As propriedades podem ter mais de uma palavra
- Neste caso precisamos colocá-las entre aspas
- Obs: não é muito utilizado, opte por camelCase

```
let carro = {
  portas: 4,
  cor: "verde",
  "tem blindagem": true
}

console.log(carro.portas);
console.log(carro["cor"]);
console.log(carro["tem blindagem"]);
```

+ {OOP}

Acessando a propriedade por variável

- Podemos acessar uma propriedade por meio de uma variável;
- É importante que a variável contenha o texto da propriedade, veja:

```
const robo = {
    bracos: 4,
    pernas: 2,
    arma: 'metralhadora',
    armaEspecial: 'foguete'
}

let a = 'arma';

console.log(robo[a]);
```

+ {OOP}

Como criar métodos

- Os métodos são as ações dos objetos;
- Podemos ter métodos de resgatar propriedades do objeto ou modificar o valor delas, por exemplo;

```
const robo = {
    bracos: 4,
    pernas: 2,
    arma: 'metralhadora',
    armaEspecial: 'foguete',
    atirar: function() {
        console.log('pew pew pew');
    }
}

robo.atirar();
```

+ {OOP}

Criando mais métodos

- Os métodos podem realizar qualquer operação que uma função realiza;

```
let pessoa = {
    nome: 'Matheus',
    getNome: function() {
        console.log("O nome é " + this.nome);
    },
    setNome: function(novoNome) {
        this.nome = novoNome;
    }
}

pessoa.getNome();
pessoa.setNome('Teste');
pessoa.getNome();
```

+ {OOP}

Objetos dentro de objetos

- Como o objeto é também um tipo de dado, podemos ter objetos com objetos dentro;
- Utilizando como um array associativo, por exemplo;

```
let pessoa = {
    nome: 'Matheus',
    caracteristicas: {
        olhos: 'verdes',
        cabelo: 'castanho',
        brincos: false,
        oculos: false
    }
}

console.log(pessoa.caracteristicas.cabelo);
```

+ {OOP}

Criando props e metodos em objs existentes

- O objeto não é imutável, ele pode ganhar propriedades e métodos ao longo do código;

```
let pessoa = {
    nome: 'Matheus',
}

pessoa.idade = 29;

pessoa.falar = function() {
    console.log('Olá');
}

console.log(pessoa);
```

+ {OOP}

Deletando propriedades e métodos

- Como é possível adicionar, também podemos deletar propriedades dos objetos;

```
let pessoa = {
    nome: 'Matheus',
}

pessoa.idade = 29;

pessoa.falar = function() {
    console.log('Olá');
}

delete pessoa.idade;
delete pessoa.falar;

console.log(pessoa);
```

+ {OOP}

Utilizando o this no objeto

- A palavra reservada **this** dentro de um objeto, vai se referir a própria instância;
- Podemos utilizar para resgatar as propriedades em um método;

```
let pessoa = {
    nome: 'Matheus',
}

pessoa.idade = 29;

pessoa.falar = function() {
    console.log('Olá, eu tenho ' + this.idade + ' anos');
}

pessoa.falar();
```

+ {OOP}

Constructor functions

- Uma outra forma de criar objeto é pela constructor function;
- Uma grande vantagem, é que este método aceita parâmetros para o obj;

```
function Pessoa(nome) {
 this.nome = nome;
}

let pessoa1 = new Pessoa('Matheus');
let pessoa2 = new Pessoa('João');

console.log(pessoa1.nome);
console.log(pessoa2.nome);
```

+ {OOP}

Funções que retornam objetos

- Parecida com as constructor functions, porém não precisamos utilizar o new;
- O objeto é criado com o retorno da função;

```
function criarPessoa(nome) {
 return {
    nome: nome
 };
}

let pessoa1 = criarPessoa('Matheus');
let pessoa2 = criarPessoa('João');

console.log(pessoa1.nome);
console.log(pessoa2.nome);
```

+ {OOP}

O objeto global

- Sempre que é iniciada uma página web traz um objeto chamado window;
- As variáveis globais são ficam atreladas a ele como propriedades;
- Os métodos da linguagem também podem ser invocados pela window;
- O this no escopo global também referenciado a window;

```
var teste = 'teste';

console.log(teste);
console.log(window.teste);
console.log(this.teste);
```

+ {OOP}

A propriedade constructor

- Quando um objeto é criado, sempre uma propriedade constructor é adicionada a ele;
- Contendo a referência de como o objeto foi criado;

```
function newObj(x) {
  return {
    x: x
  };
}

let y = newObj(1);

function NewObjTwo(x) {
  this.x = x;
}

let z = new NewObjTwo(2);

console.log(y.constructor);
console.log(z.constructor);
```

+ {OOP}

O operador instanceof

- Uma maneira de saber de qual instância (pai) vem algum objeto;
- Mais prático que utilizar o constructor;

```
function Robo(nome, arma) {
  this.nome = nome;
  this.arma = arma;
}

function Humano(nome) {
  this.nome = nome;
}

let android = new Robo('Xyz', 'Punhos');

console.log(android instanceof Robo);
console.log(android instanceof Humano);
```

+ {OOP}

Passando referência de objeto

- Quando você atribui um obj já criado para uma outra variável, você só está passando uma referência;
- Se alterar a referência, o original também é alterado;

```
let obj = {
  teste: 1,
}

let copia = obj;

copia.teste = 2;

console.log(obj.teste);
```

+ {OOP}

Comparando objetos

- Você só consegue ter objetos iguais, criando uma referência;
- Objetos criados a partir de um construtor, sempre serão diferentes;

```
function Robo(nome, arma) {
  this.nome = nome;
  this.arma = arma;
}

let robo1 = new Robo('teste', 'revolver');
let robo2 = new Robo('teste', 'revolver');

console.log(robo1 === robo2);

let robo3 = robo1;

console.log(robo1 === robo3);
```

+ {OOP}

Object literals

- Função do ES6, que permite criar objetos mais rapidamente;
- Utilizando nomes de variáveis para nomes de propriedades;

```
let x = 1;
let y = 2;

let obj = {x,y};

console.log(obj.x);
```

+ {OOP}

Object literals parte 2

- Também não precisamos declarar function para métodos no ES6;

```
let megazord = {
  nome: 'Megazord',
  arma: 'espada laser',
  explodirTudo() {
    console.log("BOOM!");
  }
}

megazord.explodirTudo();
```

+ {OOP}

Object literals parte 3

- Podemos também criar propriedades com variáveis ou retorno de funções;
- Ajudando a escrever menos código;

```
let tipo = 'tipo_de_';

let carro = {
  [tipo+"carro"]: "SUV"
}

let barco = {
  [tipo="barco"]: "Iate"
}

console.log(carro.tipo_de_carro);
```

+ {OOP}

Atributos das propriedades

- Toda propriedade tem atributos já criados pela linguagem, Enumerable e Configurable;
- Configurable, por exemplo, se estiver false, não deixa a propriedade ser editada ou deletada;

```
let pessoa = {
  nome: "Matheus"
}

console.log(Object.getOwnPropertyDescriptor(pessoa, 'nome'));
```

{OOP}

Copiando propriedades

- Os objetos herdam métodos do objeto pai Object, e podemos utilizá-los;
- Para copiar propriedades utilizamos o método **assign**

```
let robo1 = {
  arma: 'lança granada'
}

let robo2 = {

}

Object.assign(robo2, robo1);

console.log(robo2);
```

+ {OOP}

Comparando objetos

- Podemos comparar os objetos com o método **is**
- Teremos basicamente os mesmos resultados de ===
- Salvo para NaN com NaN e +0 com -0 (que neste método são considerados como iguais)

```
console.log(Object.is(robo1, robo2));

robo3 = robo1;

console.log(Object.is(robo1, robo3));
```

+ {OOP}

Destructuring

- Um outro recurso que veio com o ES6 trazendo algumas funcionalidades;
- Podemos criar várias variáveis com uma linha só, desestruturando um objeto;

```
let config = {
  ip: '127.0.0.1',
  port: '80',
  blocked: true,
}

let {ip, port, blocked} = config;

console.log(ip);
console.log(port);
console.log(blocked);
```

+ {OOP}

Destructuring parte 2

- Há também a possibilidade de utilizar o destructuring para mudar o valor de variáveis já criadas;

```
let config = {
  ip: '127.0.0.1',
  port: '80',
  blocked: true,
}

let ip = '192.168.0.60';
let port = '8000';

({ip, port} = config);

console.log(ip, port);
```

+ {OOP}

Destructuring parte 3

- O destructuring funciona com arrays também

```
let numeros = [1,2,3];

const [num1,num2,num3] = numeros;

console.log(num1,num2,num3);
```

+ {OOP}

Destructuring e rest operator

- O rest operator é utilizado quando não sabemos quantos argumentos virão para o destructuring;
- Podendo criar um array com um tamanho infinito, com os restos dos elementos passados;

```
let [a, ...b] = [1,2,3,4,5,6,7,8];

console.log(a, b);
```

+ {OOP}

Objetos 02

Conclusão da unidade

+ {OOP}

Exercícios de Objetos

03

+ {OOP}

Exercício sobre os conceitos de objeto aprendidos anteriormente

Exercício 1

- Crie um objeto com 3 propriedades;
- A primeira deve ser uma string, a segunda um number e a terceira um boolean;

+ {OOP}

Exercício 2

- Crie um objeto Pessoa, que tem uma propriedade nome;
- Crie um método que exibe o nome do objeto Pessoa;

+ {OOP}

Exercício 3

- Crie um objeto Ninja, por constructor function;
- Com a propriedade de nome do ninja e o método atirarShuriken;

+ {OOP}

Exercício 4

- No objeto ninja que a propriedade shuriken, com uma quantidade de estrelas ninjas;
- A cada método do disparo subtraia uma;
- O ninja não pode jogar mais shurikens caso elas tenham acabado;

+ {OOP}

Exercício 5

- Crie um objeto Inimigo, com as propriedades nome e vivo (que é um boolean e inicie como true);
- O método atirarShuriken do exercício passado deve ‘matar’ o Inimigo, setando a propriedade do Inimigo vivo como false;

+ {OOP}

Exercício 6

- Crie uma função que retorna se o objeto é uma instância de outro objeto;
- Deve apresentar no console as mensagens de positivo e negativo;

+ {OOP}

Exercício 7

- Crie dois objetos que compartilhem nomes de propriedades via object literals;
- Uma variável deve definir a parte que se repete nas propriedades dos objetos;

+ {OOP}

Exercício 8

- Crie um objeto que tenha características de um caminhão e coloque em propriedades distintas;
- Com destructuring coloque essas propriedades em variáveis separadas;

+ {OOP}

Exercícios de Objetos

03

Conclusão da unidade

+ {OOP}

Principais conceitos de OO

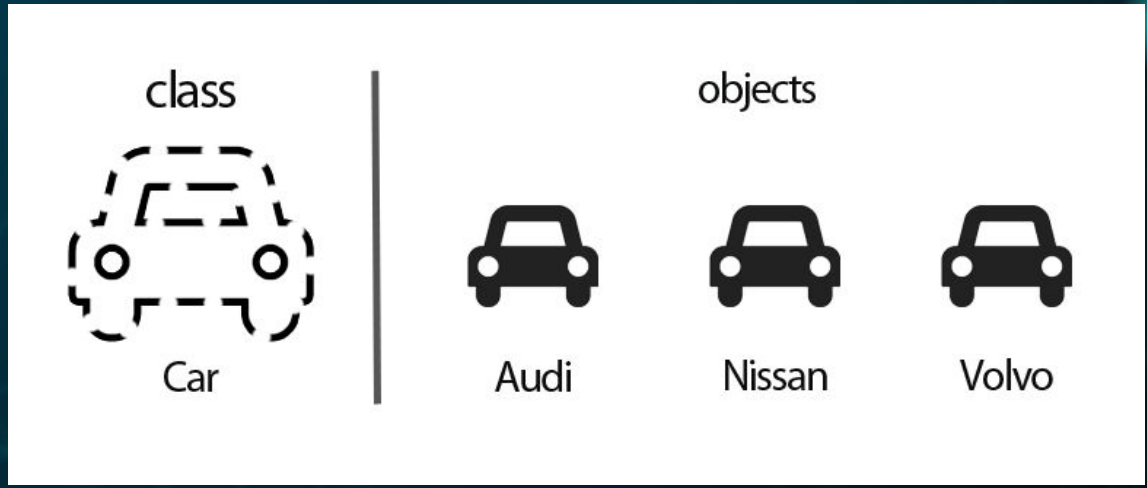
+ {OOP}

04

Teoria sobre os principais conceitos da orientação a objetos

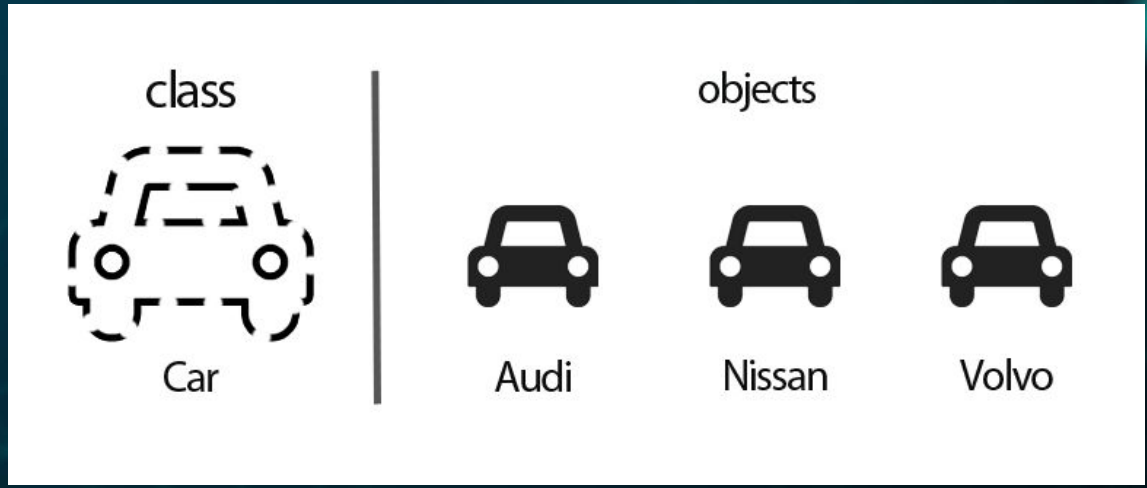
Objetos

- A representação de uma Classe;
- Fundamental para a Orientação a Objetos;
- Tem propriedades que representam as características de um objeto;
- Tem métodos que representam as ações de um objeto;



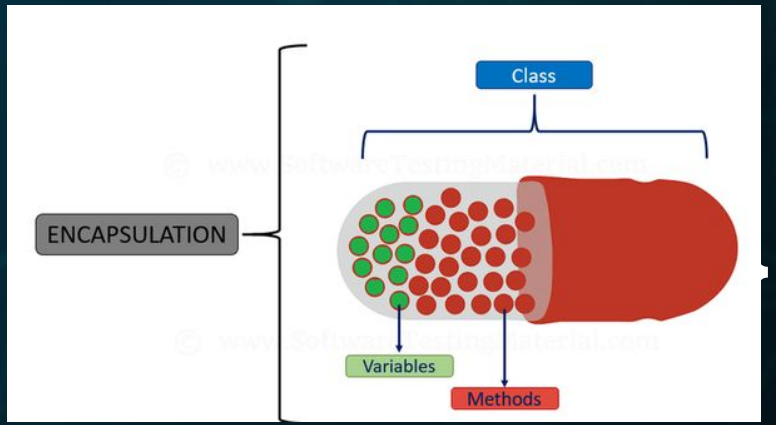
Classes

- É como se fosse o molde do objeto;
- Geralmente se criam diversos objetos da mesma classe;
- Normalmente já possui as propriedades e métodos que os objetos vão utilizar;
- A Classe no JS foi introduzida na versão ES6 da linguagem;



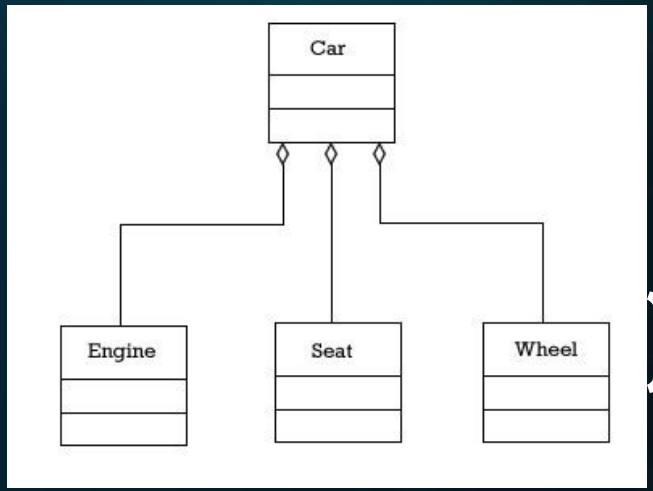
Encapsulação (Encapsulation)

- Quando um objeto contém, ou encapsula, dados ou meios de fazer algo com os dados (usando métodos);
- Um outro aspecto da encapsulação é ter propriedades e métodos: **públicos**, **privados** ou **protegidos;**
- No JS não temos estes meios de forma nativa, tudo é público;
- Porém podemos contornar isso;



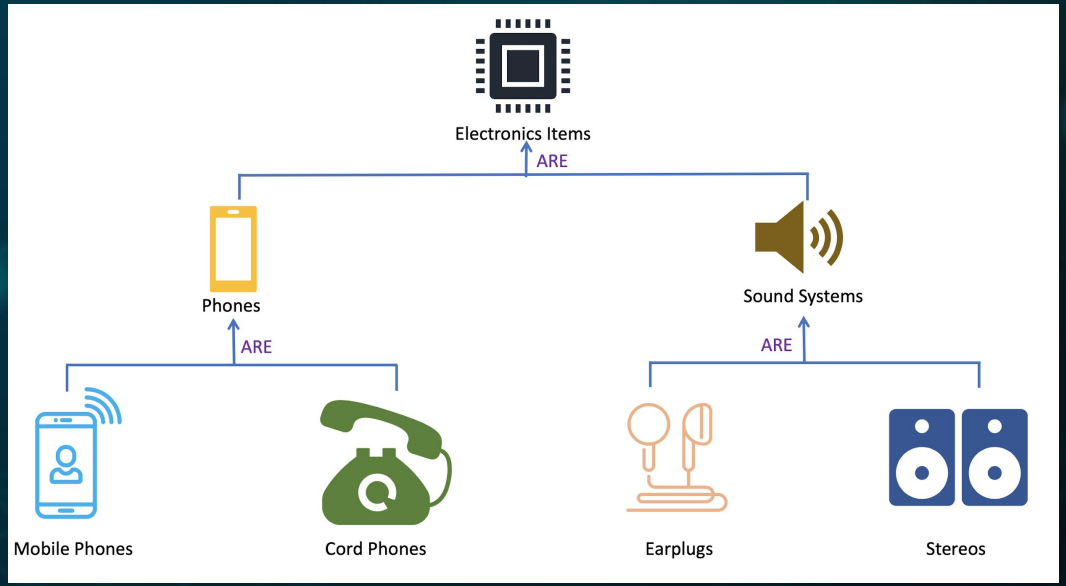
Agregação (Aggregation)

- Também conhecido como Composição (composition);
- O ato de combinar diversos objetos em um maior;
- Isso serve para não termos um objeto muito grande e complexo;
- Objeto grande = SalaDeAula;
- Objeto com Aggregation = SalaDeAula com Aluno, Cadeira, Lousa, e etc.
- A sala de aula foi dividida em diversos objetos, que cada um tem sua responsabilidade;



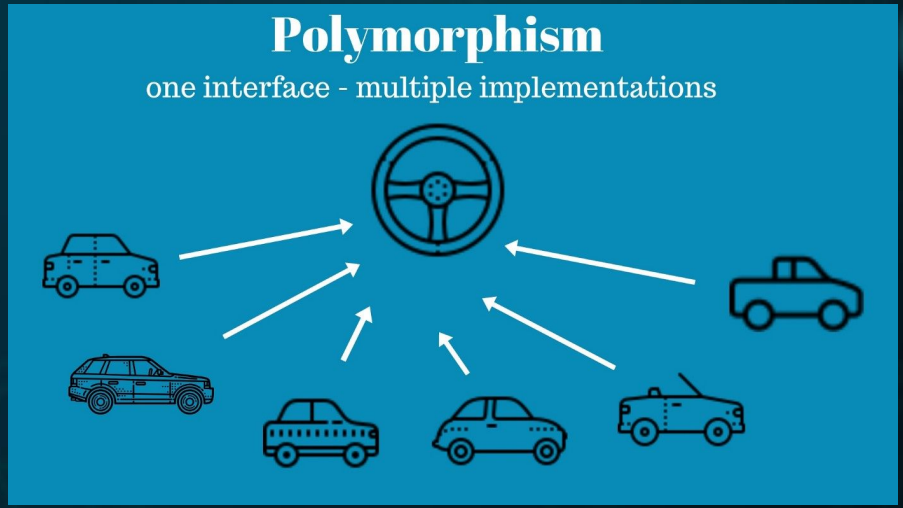
Herança (Inheritance)

- Quando um objeto ou classe deriva de uma outra classe;
- Podendo herdar suas propriedades e métodos;
- Fazendo com que você crie classes com o comportamento semelhante, porém para fins distintos;



Polimorfismo (Polymorphism)

- É a possibilidade de utilizar um método de uma classe pai de uma maneira diferente;
- Que se adapte as necessidades do novo objeto, sem precisar alterar o método do objeto pai;
- Importante citar que o polimorfismo utiliza o conceito de herança;



Principais conceitos de OO

04

Conclusão da unidade

+ {OOP}

objetos do JavaScript

05

+ {OOP}

Conhecer os objetos que foram desenvolvidas para a linguagem

Os built in objects

- Objetos que são criados pelos desenvolvedores da linguagem;
- Que originam strings, arrays e objetos;
- Este objetos criados por nós herdam suas propriedades e métodos;
- Um aspecto da **herança;**
- Porém no JavaScript isto é mais conhecido pelo conceito de **Prototype**, que todo objeto tem um pai;

+ {OOP}

Object

- O pai de todos os objetos do JavaScript;
- Possui propriedades e métodos, mesmo o objeto estando vazio;
- Pode ser criado via new;

```
let o = new Object();

console.log(o.toString()); // representação do objeto em string

console.log(o.valueOf()) // retorna o próprio objeto
```

+ {OOP}

Array

- O objeto pai de todos os arrays;
- Pode instanciar um array com new;
- Possui também propriedades e métodos;

```
let a = new Array(1,2,3);
a[3] = 4;

a.toString();

console.log(Array instanceof Object); // adivinha a resposta :D
```

+ {OOP}

Array parte 2

- A propriedade length indica o número itens de um array;
- E temos métodos famosos como: **push**, **pop** e **join**;
- É de extrema importância conhecer os métodos de arrays para programar bem em JS;

```
let a = [];

a.push('elemento');

console.log(a);

a.pop(); // remove o último elemento

console.log(a);
```

+ {OOP}

Function

- O objeto para criar funções;
- Podemos criar novas funções a partir de new;
- Obs: não é utilizado, serve apenas para conhecimento e para você entender como o JS funciona;

```
let teste = new Function(
  'a',
  'return arguments'
);

console.log(teste('testando Function'));
```

+ {OOP}

Function parte 2

- Podemos utilizar a propriedade length para saber o número de argumentos de uma função
- Temos também o método toString neste objeto;

```
function test(a,b) {
  return a + b;
}

console.log(test.length);
```

+ {OOP}

Function parte 3

- Os métodos que podemos utilizar do Function são call e apply;
- O call pode pegar métodos emprestado de objetos;
- O método apply funciona igual o call, mas todos os parâmetros são transformados em arrays;

```
let a = {
  name: "A",
  falar() {
    console.log("Olá sou o método do " + this.name);
  }
}

b = {
  name: "B",
};

a.falar.call(b);
```

+ {OOP}

Boolean

- O Boolean também é um objeto e serve para valores booleanos (true e false);
- Podemos criar com new e o método valueOf() da o valor do booleano;
- Este objeto não tem métodos;
- E é claro, você pode dispensar a criação de um boolean com o objeto, utilize o método convencional;

```
let x = new Boolean(false);

console.log(x.valueOf());
```

+ {OOP}

Number

- O Number também um objeto para tratar os números, tem métodos conhecidos como **parseInt** e **parseFloat;**
- Podemos criar um novo objeto com new também;

```
console.log(Number.parseInt(12.7327)); // 12

console.log(parseInt(12.7327)); // 12

console.log(Number.parseFloat('12.83')) // 12.83
```

+ {OOP}

Number parte 2

- Algumas propriedades que são interessantes no Number são MAX_VALUE e MIN_VALUE, para saber o máximo e o mínimo que o JS atinge;

```
console.log(Number.MAX_VALUE);

console.log(Number.MIN_VALUE);

console.log(Number.NaN);
```

+ {OOP}

Number parte 3

- Alguns outros métodos importantes de Number são: toFixed, toPrecision e toExponential;
- E o detalhe é que não precisamos utilizar estes métodos com o Number, só o método já será interpretado pelo JS;

```
console.log((123.3448384).toFixed(2));

console.log((124.34).toFixed());

console.log((1000).toExponential());
```

+ {OOP}

String

- Podemos criar uma string em objeto com o new;
- Temos acesso a propriedade length, que dá o número de caracteres;
- Podemos acessar um caractere pelo seu índice;

```
let stringObj = new String('teste');
let string = 'teste';

console.log(stringObj.length);
console.log(string.length);

console.log(stringObj[1]);
console.log(string[1]);
```

+ {OOP}

String parte 2

- O objeto String também dá muitos métodos interessantes;
- E como os arrays é de suma importância conhecer estes métodos;

```
let string = 'teste';

console.log(string.toUpperCase());
console.log(("ASD").toLowerCase());
console.log(string.charAt(4));
console.log(string.indexOf('s'));
```

+ {OOP}

String parte 3

- Conheça mais alguns métodos de string;

```
let string = 'teste';

console.log(string.slice(1,3));
console.log(string.substring(1,3));
console.log(string.split(""));
```

+ {OOP}

Math

- Um objeto com propriedades e métodos matemáticos;
- Podemos saber o valor de PI e até gerar números aleatórios;

```
console.log(Math.PI);
console.log(Math.LN2)
console.log(Math.random());
```

+ {OOP}

Math parte 2

- Temos também métodos de arredondamento como: floor, ceil e round;
- Métodos para calcular potência;
- E também raiz quadrada;

```
console.log(Math.floor(123.22));
console.log(Math.pow(12,2));
console.log(Math.sqrt(9));
```

+ {OOP}

Date

- Objeto que lida com datas;
- Podemos criar uma nova data a partir de agora;
- Ou a partir de uma data que precisarmos;

```
console.log(new Date());
console.log(new Date(2020,4,30));
console.log(new Date(1357027200000));
```

+ {OOP}

Date parte 2

- Temos diversos métodos para aplicar em datas;

```
let date = new Date();

console.log(date.setMonth(5));

console.log(Date.parse('Apr 22, 2019'));

console.log(Date.now());

console.log(new Date(Date.now()));
```

+ {OOP}

RegExp

- Objeto para tratar expressões regulares;
- Podemos utilizar métodos como test e exec;

```
let regex = new RegExp(/t/);

console.log(regex.test("teste"));
console.log(regex.test("opa"));
```

+ {OOP}

Error

- Objeto para tratar de erros;
- Ele é o que deriva dos erros que recebemos;
- E também podemos criar os nossos erros;

```
try {
  throw new Error('Deu errado!');
} catch(e) {
  console.log(e.name + ": " + e.message);
}
```

+ {OOP}

objetos do JavaScript

05

Conclusão da unidade

+ {OOP}

Prototypes

06

+ {OOP}

Vamos aprender para que servem e como utilizar os Prototypes

O que são Prototypes?

- JavaScript é uma linguagem considerada baseada em prototypes;
- Todos os objetos do JS herdam propriedades e métodos do seu Prototype;
- Como vimos nos casos dos built in objects;
- **A ideia central é que:** todo objeto tenha um pai (ou seja, um Prototype);

+ {OOP}

A propriedade prototype

- As funções além de suas propriedades que já vimos, também vem com a propriedade **prototype** criada;
- Recebemos um objeto vazio, que pode ter propriedades e métodos adicionados;

```
function test() {return true};

console.log(test.prototype);
console.log(typeof test.prototype);
```

+ {OOP}

Adicionando props e métodos com prototype

- Vejamos agora como podemos adicionar propriedades e métodos;
- Perceba que não há diferença em acessá-las;

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

Pessoa.prototype.profissao = 'Estudante';
Pessoa.prototype.falar = function() {
  console.log("Olá mundo!");
}

let joao = new Pessoa("João", 15);

joao.falar();
console.log(joao.profissao);
```

+ {OOP}

Adicionando múltiplas props e métodos

- Não precisamos adicionar uma a uma as propriedades ou métodos;

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

Pessoa.prototype = {
  profissao: 'Estudante',
  falar() {
    console.log("Olá mundo!");
  }
}

let joao = new Pessoa("João", 15);

joao.falar();
console.log(joao.profissao);
```

+ {OOP}

Modificação do prototype

- Ao alterar o prototype, todas as instâncias ganham seus novos métodos ou propriedades;

```
let joao = new Pessoa("João", 15);

Pessoa.prototype.gritar = function() {
  console.log("AHHHHHHHH");
}

joao.gritar();
```

+ {OOP}

Props do obj x props do Prototype

- A ordem de acesso é: primeiro o objeto e depois o prototype;
- As propriedades podem coexistir;

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

Pessoa.prototype.idade = 10;
Pessoa.prototype.cabelo = 'castanho';

let pedro = new Pessoa("Pedro", 15);

console.log(pedro.idade);
console.log(pedro.cabelo);

pedro.cabelo = 'louro';

console.log(pedro.cabelo);
```

+ {OOP}

Maneira de utilizar o prototype se já tem prop

- Podemos deletar uma propriedade, e voltar a utilizar o prototype;
- Pois mesmo sendo sobrescrito, ainda estará disponível;

```
function Pessoa(name) {
    this.name = name
}

Pessoa.prototype.name = 'estava sobrescrito';

let pessoa = new Pessoa('teste');

console.log(pessoa.name)

delete pessoa.name;

console.log(pessoa.name);
```

+ {OOP}

Verificando propriedade do prototype

- Da mesma forma que o objeto tem método de verificar propriedades;
- Os prototypes tem também;

```
function Pessoa(name) {
    this.name = name
}

Pessoa.prototype.name = 'estava sobrescrito';

let pessoa = new Pessoa('teste');

console.log(pessoa.hasOwnProperty('name'));
console.log(pessoa.constructor.prototype.hasOwnProperty('name'));
```

+ {OOP}

Distinguir se é prop do obj ou do prototype

- E é claro que utilizando o método hasOwnProperty, conseguimos saber se a propriedade é do objeto ou do prototype;

```
function Pessoa(name, lastname, age) {
    this.name = name;
    this.lastname = lastname;
    this.age = age;
}

Pessoa.prototype.job = 'programador';

let p = new Pessoa("João", "Silva", 40);

console.log(p.hasOwnProperty("job"));
console.log(p.constructor.prototype.hasOwnProperty("job"));
```

+ {OOP}

Loop para objetos

- **Obs:** um pouco off topic de prototype :D
- O loop mais indicado para percorrer objetos é o for ... in;
- Isso por que o for normal serve mais para arrays;

```
function Pessoa(name, lastname, age) {
    this.name = name;
    this.lastname = lastname;
    this.age = age;
}

pessoa = new Pessoa('Matheus', 'Battisti', 29);

for(prop in pessoa) {
    console.log(prop + " -> " + pessoa[prop]);
}
```

+ {OOP}

Checar se é prototype de algum objeto

- Com o método isPrototypeOf, conseguimos checar se um objeto é prototype de outro;

```
let caracteristicas = {
    maos: 2,
    pes: 2,
    pernas: 2
}

function Humano(name) {
    this.name = name;
}

Humano.prototype = caracteristicas;

let joao = new Humano("João");

console.log(caracteristicas.isPrototypeOf(joao));
```

+ {OOP}

Melhorando os objetos do JavaScript

- Podemos criar novos métodos e propriedades para os objetos do JS existentes;

```
Array.prototype.checkLength = function() {
    return this.length;
}


let a = [1,2,3];

console.log(Array.checkLength());
```

+ {OOP}

Porque não é uma excelente ideia

- Futuramente pode vir um método com o mesmo nome na linguagem;
- Desenvolvedores podem desconhecer o método, não sabendo de onde ele vem, gerando confusão;
- Para minimizar os problemas, faça uma checagem antes;

```
if(typeof Array.prototype.checkLength !== 'function') {
    Array.prototype.checkLength = function() {
        return this.length;
    }
}
```

+ {OOP}

Prototypes 06

Conclusão da unidade

+ {OOP}

Herança

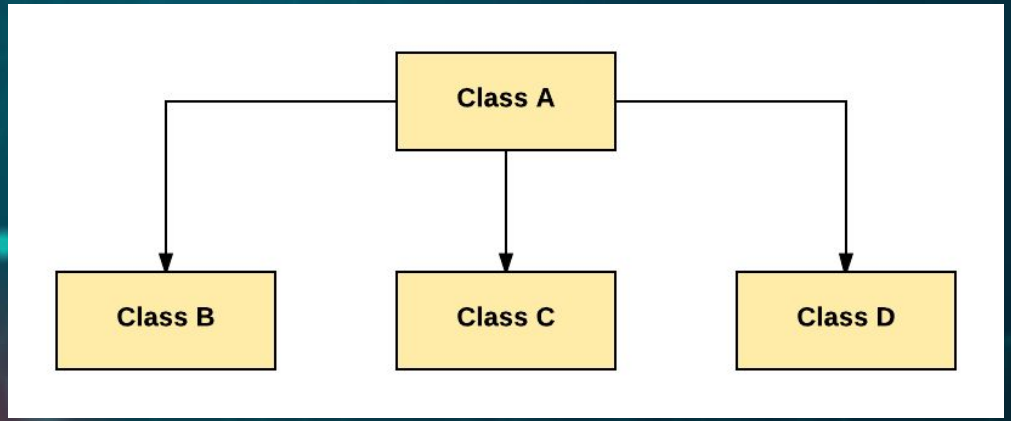
07

+ {OOP}

Vamos aprender como utilizar o conceito de herança no JS

A herança e o JavaScript

- Reutilzação de código;
- As propriedades e métodos são passadas para outros objetos filhos;
- A herança do JS pode ser aplicada via prototype chain;
- Porém há outras maneiras de atingir este objetivo (veremos nesta seção);



+ {OOP}

Prototype chain

- É maneira default da linguagem de fazer herança;
- Podemos instanciar objetos no prototype de outros, criando a herança;

```
function Pessoa() {
    this.classe = 'Mamífero';
    this.falar = function() {
        console.log("Olá");
    }
}

function Advogado() {
    this.profissao = 'Advogado';
}

Advogado.prototype = new Pessoa();

let joao = new Advogado();

joao.falar();
```

+ {OOP}

Checando a herança

- Quando utilizamos a prototype chain, o objeto passa a virar instância de todos os ‘pais’;
- Podemos verificar isso pela instrução instanceof;

```
console.log(joao instanceof Advogado);
console.log(joao instanceof Pessoa);
console.log(joao instanceof Object);
```

+ {OOP}

Métodos e props no Prototype

- A ideia de utilizar o prototype é para que cada prop ou método adicionado nele não se repita a cada objeto instanciado;
- Então esta herança beneficia o código, criando uma referência para os novos objetos, deixando o programa mais performático;
- Não ocupando um novo espaço na memória a cada obj criado;

```
function Pessoa() {}

Pessoa.prototype.classe = 'Mamífero';
Pessoa.prototype.falar = function() {
    console.log("Olá");
}

function Advogado() {}

Advogado.prototype.profissao = 'Advogado';

Advogado.prototype = new Pessoa();

let joao = new Advogado();

joao.falar();
```

+ {OOP}

Aumentando ainda mais a eficiência

- Vimos que utilizar o prototype é uma boa prática;
- Então por que não clonar só o prototype em vez da instância do objeto?

```
function Pessoa() {}

Pessoa.prototype.classe = 'Mamífero';
Pessoa.prototype.falar = function() {
    console.log("Olá");
}

function Advogado() {}

Advogado.prototype.profissao = 'Advogado';

// clonando apenas o prototype de Pessoa
Advogado.prototype = Pessoa.prototype;

let joao = new Advogado();

joao.falar();
```

+ {OOP}

Precauções

- Utilizando a abordagem de clonar só o prototype tem um side effect;
- Se você muda o prototype, toda a cadeia que o utiliza, vai ser alterada também;
- Então utilize desse jeito apenas quando não precisa mudar métodos e propriedades;

```
let joao = new Advogado();

Advogado.prototype.falar = function() {
    console.log("Tchau");
}

let pedro = new Pessoa();

pedro.falar();
```

+ {OOP}

Construtor temporário

- Caso você tenha uma solução que não te deixaria optar por propriedades e métodos que não são alteráveis, você pode utilizar um construtor temporário e resolver o problema;

```
// clonando apenas o prototype de Pessoa, com construtor temporário
let F = function() {};
F.prototype = Pessoa.prototype;
Advogado.prototype = new F();
```

+ {OOP}

Isolando a herança em uma função

- Para facilitar as coisas e deixar a herança reutilizável também, podemos utilizar uma função;

```
function extend(Filho, Pai) {

    let F = function() {};
    F.prototype = Pai.prototype;
    Filho.prototype = new F();

}

function Pessoa() {}

Pessoa.prototype.classe = 'Mamífero';
Pessoa.prototype.falar = function() {
    console.log("Olá");
}

function Advogado() {}

Advogado.prototype.profissao = 'Advogado';

// herança
extend(Advogado, Pessoa)

let joao = new Advogado;

joao.falar();
```

Copiando propriedades

- Podemos em vez de utilizar o fake constructor copiar as propriedades por um loop e realizar a herança;
- Precisamos utilizar a propriedade uber, que nos dará acesso ao obj Pai;
- A parte ruim desta abordagem é que ela recria as propriedades e métodos;

```
function extend(Filho, Pai) {
    let paiProto = Pai.prototype;
    let filhoProto = Filho.prototype;
    for(let i in paiProto) {
        filhoProto[i] = paiProto[i];
    }
    // filho tem acesso ao obj pai
    filhoProto.uber = paiProto;
}

function Veiculo() {}

Veiculo.prototype.motor = 1;
Veiculo.prototype.carenagem = 'aço'

function Carro(cor) {
    this.cor = cor;
}

Carro.prototype.portas = 4;

extend(Carro, Veiculo);

let bmw = new Carro('azul');

console.log(bmw.carenagem);
```

Outro problema ao copiar por loop

- Os arrays ficam alocados na memória e é criado apenas uma referência, fazendo com o que se o array do filho se altere o do pai também;

```
Veiculo.prototype.opicionais = ['teto solar', 'aro de alumínio', 'diplay 8"'];

extend(Carro, Veiculo);

console.log(Veiculo.prototype);

Carro.prototype.opicionais.push('blindagem');

console.log(Veiculo.prototype.opicionais);
```

Resolvendo o problema

- Podemos utilizar uma estrategia de copiar um objeto, resolvendo este problema;
- Porém veja que o código fica complexo demais, talvez não seja o caso de utilizar herança para isso;
- Além de não utilizar prototypes nesta forma;

```
function objectClone(o, stuff) {
  var n;
  function F() {}
  F.prototype = o;
  n = new F();
  n.uber = o;
  for (var i in stuff) {
   n[i] = stuff[i];
  }
  return n;
}


function Veiculo() {
  this.carenagem = 'aço';
  this.opicionais = ['blindagem', 'lanterna LED'];
}

let v = new Veiculo;

let ferrari = objectClone(v, {
    rodas: 4,
  marca: 'Ferarri'
});

console.log(ferrari);
```

Herança múltipla

- Uma estrutura difícil de manter e o JS não nos dá esta funcionalidade de forma fácil, precisamos criar a função;
- É difícil de manter;
- Melhor optar por prototype chain;

```
function multi() {
  var n = {}, stuff, j = 0, len = arguments.length;
  for (j = 0; j <len; j++) {
    stuff = arguments[j];
    for (var i in stuff) {
      if (stuff.hasOwnProperty(i)) {
        n[i] = stuff[i];
      }
    }
  }
  return n;
}

let pneu = {
    material: 'borracha'
}

let aro = {
    tamanho: 20
}

let pneuMontado = multi(pneu, aro);

console.log(pneuMontado);
```

Herança 07

Conclusão da unidade

+ {OOP}

Classes ES6

08

+ {OOP}

Veremos o poder do ES6 aos recursos que competem a classes e módulos

Classes no JavaScript

- As classes na verdade são funções, ou seja, muda a forma de nós escrevermos mas o JS utiliza as mesmas técnicas que utilizamos antes;
- Tornando este método um **syntatic sugar**;
- Então aprender como funciona por baixo dos panos, como foi visto ao longo do curso, nos ajudará a entender estes conceitos novos de forma mais fácil;

+ {OOP}

Definindo classes

- A declaração é bem parecida com constructor functions;
- As propriedades devem ficar num método especial chamado **constructor**;
- Onde serão inicializadas;

```
class Tenis {
    constructor(modelo, cor) {
        this.modelo = modelo;
        this.cor = cor;
    }
}

console.log(typeof Tenis);

let allstar = new Tenis("All Star", "Branco");
```

+ {OOP}

Outra maneira de criar classes

- Outra maneira de criar classes é a conhecida como classe anônima;

```
let Tenis = class {
    constructor(modelo, cor) {
        this.modelo = modelo;
        this.cor = cor;
    };
  modeloDoTenis() {
    return this.modelo;
  }
}

let allstar = new Tenis('All Star', 'preto');

console.log(allstar);
```

+ {OOP}

Curiosidades sobre o constructor

- É utilizado apenas para inicializar valores de propriedades;
- Você só pode utilizar um constructor por classe;
- O constructor pode chamar a classe pai por uma instrução super (o que ajuda na herança);

+ {OOP}

Prototype methods

- São métodos que já existem na Class, por exemplo os getters e setters;

```
let Tenis = class {
    constructor(modelo, cor) {
        this.modelo = modelo;
        this.cor = cor;
    }
    modeloDoTenis() {
        return this.modelo;
    }
    set trocarModelo(novoModelo) {
        this.modelo = novoModelo;
    }
    get obterModelo() {
        return "O modelo do tênis é :" + this.modelo;
    }
}

let allstar = new Tenis('All Star', 'preto');

allstar.trocarModelo = "All Star 2.0";
console.log(allstar.obterModelo);
```

+ {OOP}

Métodos estáticos

- Métodos que só funcionam caso você utilize o nome da classe;
- Ou seja, você não cria um novo objeto para usar eles;
- Utilizado muito como um helper;

```
class Calc {
    static soma(a, b) {
        return a + b;
    }
}

console.log(Calc.soma(2,5));
```

+ {OOP}

Subclasses

- Uma forma de criar herança com as classes;
- Utilizando a palavra extends, uma classe herda as propriedades de outra;
- Bem mais fácil, não? :D

```
class Animal {
    constructor(nome) {
        this.nome = nome;
    }
}

class Cachorro extends Animal {
    latir() {
        console.log("Au au");
    }
}

let bob = new Cachorro("Bob");

bob.latir();

console.log(bob.nome);
```

+ {OOP}

Classes ES6

08

Conclusão da unidade

+ {OOP}

TypeScript

09

+ {OOP}

Vamos aprender os conceitos fundamentais do TypeScript e como aplicá-los

O que é TypeScript?

- É um superset para a linguagem JavaScript;
- Que alguns consideram uma linguagem;
- Traz algumas funcionalidades novas como por exemplo: Types, features de versões futuras do JavaScript, Interfaces, Generics e Decorators;
- Nós não executamos o TS, compilamos em JS e utilizamos a versão gerada de JS;
- Utiliza-se em conjunto de outras tecnologias, como: React;

+ {OOP}

Como instalar o TypeScript

- Primeiramente é necessário instalar o Node;
- Para instalar: npm install -g typescript
- Pronto! Agora você pode compilar arquivos de TS em JS;
- É indicado também o uso de IDEs que façam highlight de sintaxe do TS, como: VS Code

Testando e compilando o TS

- Iniciaremos criando uma aplicação em HTML normal;
- Porém com um arquivo .ts que será o nosso arquivo fonte de JavaScript futuramente;
- Após o código finalizado, podemos compilar e testar com o comando:
- **tsc arquivo.ts**

Tipos de dados básicos

- No TypeScript temos alguns tipos semelhantes ao JavaScript, como: boolean, number e string;
- A vantagem do TS é o type cast, garantindo que a variável contenha um valor daquele tipo:

```
const numero: number = 2;
const nome: string = 'Matheus';
const isProgrammer: boolean = true;
```

+ {OOP}

Arrays e Objetos

- Também podemos representar arrays e objetos pelo TS, veja:

```ts
// primeira opção
const lista: number[] = [1, 2, 3];
// segunda opção
const lista2: Array<string> = ['João', 'Maria', 'Matheus'];

const a = { nome: 'Matheus', idade: 29 }
```

+ {OOP}

Outros tipos de dados

- O TS nos provê outros tipos de dados, que são: tuplas, Enum, Any, Void, Null, Undefined e Never;
- A tupla serve para determinamos um conjunto de valores fixos, veja:

```
// Declarando uma tupla
let pessoa: [string, number];

pessoa = ['João', 20];
```

Outros tipos de dados

- O Enum nos dá a possibilidade de criar um conjunto apenas com valores numéricos, veja:

```
enum Carro { Motor = 1, Portas = 4, Pedais = 3 }

let numeroDeMotores = Carro.Motor;

console.log(Carro);
console.log(numeroDeMotores);
```

Outros tipos de dados

- O Any nos possibilita inserir uma variável com qualquer tipo de dado;
- Devemos utilizar em casos extremamente necessários, pois vai contra a ideia de ‘tipar’ variáveis;

```
let aindaNaoSabemos: any;

aindaNaoSabemos = 1;
aindaNaoSabemos = 'teste';
aindaNaoSabemos = true;
```

Outros tipos de dados

- O Void é o oposto de Any, geralmente declarado em funções, pois em variáveis não tem muita utilidade, já que aceita apenas undefined;

```
function logMessage(): void {
  console.log("Mensagem do sistema!");
}

logMessage();
```

Interfaces

- Interface é a possibilidade de criar uma função que recebe argumentos específicos e os utiliza conforme usa lógica, veja:

```
function imprimirNome(obj: { nome: string }) {
  console.log(obj.nome);
}

let pessoa = { nome: 'Matheus', idade: 29 };
imprimirNome(pessoa);
```

+ {OOP}

Mais sobre Interfaces

- Podemos criar interfaces com parâmetros opcionais também:

```
interface config {
    nome?: string;
    idade?: number;
}

function criarPessoa(config: config): {nome: string; idade: number} {
    let pessoa = { nome: "Não informado", idade: 0 }

    if (config.nome) {
        pessoa.nome = config.nome;
    }
    if (config.idade) {
        pessoa.idade = config.idade;
    }

    return pessoa;
}

let joao = criarPessoa({ nome: "João" });

console.log(joao);
```

+ {OOP}

Classes

- Uma outra grande funcionalidade do TS são as classes, que utilizam a ideia da versão ES6 do JS e não a herança pro prototype;

```
class Carro {
  marca: string;
  aro: number;
  constructor(marca: string, aro: number) {
      this.marca = marca;
      this.aro = aro;
  }
  verificarAro() {
      return `O aro do carro é: ${this.aro}`;
  }
}

let ferrari = new Carro("Ferrari", 20);
console.log(ferrari);
```

+ {OOP}

Herança

- A herança do TS também lembra muito a herança do ES6:

```
class Animal {
  andar() {
      console.log("O animou andou");
  }
}

class Cachorro extends Animal {
  nome: string;
  constructor(nome: string) {
      super();
      this.nome = nome;
  }
  latir() {
      console.log("Au Au");
  }
}
```

+ {OOP}

Funções

- Podemos adicionar os tipos a parâmetros de funções também, assim nosso código ficará mais seguro:

```
function somar(x: number, y: number): number {
  return x + y;
}

console.log(somar(5, 6)); // 11
console.log(somar(5, '5')); // 55
```

+ {OOP}

Mais sobre Funções

- As funções também aceitam parâmetros opcionais, veja:

```
function bemVindo(saudacao?: string, nome: string) {
    if (saudacao) {
        console.log(`Olá ${saudacao} ${nome}`);
    } else {
        console.log(`Olá ${nome}`);
    }
}

bemVindo("Sr.", "Matheus");
bemVindo(undefined, "Lucas" );
```

+ {OOP}

Generics

- Uma forma de criar componentes sem um tipo específico, com a possibilidade de reutilização;
- Cria-se um placeholder de um type, que será preenchido na execução;

```
function identity<T>(arg: T): T {
    console.log(typeof arg)
    return arg;
}

console.log(identity('asd'));
console.log(identity(1));
```

Mais sobre Generics

- Você pode determinar o tipo ao invocar a função:

```ts
function identity<T>(arg: T): T {
  console.log(typeof arg)
  return arg;
}

console.log(identity<number>('asd')); // sinalizacao de erro
console.log(identity<number>(1));
```

TypeScript 09

Conclusão da unidade

+ {OOP}