

INSTITUTO FEDERAL DE MATO GROSSO DO SUL
CAMPUS NOVA ANDRADINA

NOTAS DE AULA

LINGUAGEM E TÉCNICAS DE
PROGRAMAÇÃO

Prof. Me. Luiz F. Picolo

NOVA ANDRADINA - MS

Atualizado em 7 de fevereiro de 2023

1 Introdução

*Persistência é a irmã gêmea da excelência.
Uma é a mãe da qualidade, a outra é a mãe do tempo.*

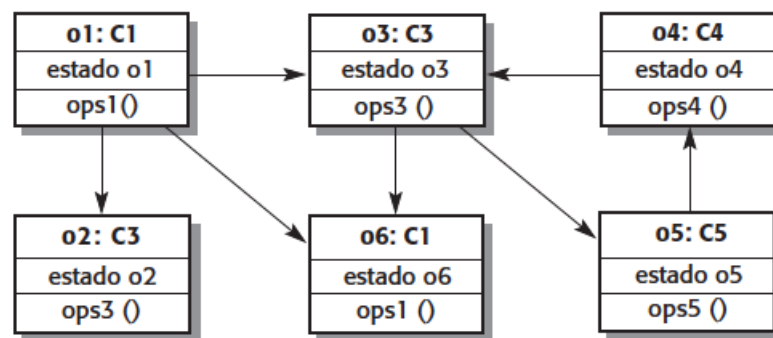
Marabel Morgan

Antes de prosseguir com a disciplina, é importante que tenhamos uma compreensão clara e detalhada dos conceitos relacionados à **Orientação a Objetos (POO)**. Esses conceitos incluem **Classes, Métodos, Herança, Objetos** e outros conceitos interligados. A compreensão desses conceitos é fundamental para podermos aproveitar ao máximo as vantagens do paradigma da Orientação a Objetos e utilizá-lo de maneira efetiva na solução de problemas complexos. É através da compreensão desses conceitos que poderemos desenvolver soluções mais organizadas, claras e eficientes. Por isso, este capítulo introdutório visa apresentar esses conceitos de concisamente, proporcionando uma base sólida para o desenvolvimento de habilidades e conhecimentos mais avançados no decorrer da disciplina. Então, estejam atentos e preparem-se para mergulhar no mundo da Orientação a Objetos.

1.1 Orientação a objeto

O Paradigma da Orientada a Objetos (também conhecida pela sua sigla POO) pretende representar o mais fielmente possível as situações reais nos sistemas computacionais. Nós entendemos o mundo todo composto por vários objetos que interagem uns com os outros. Da mesma maneira, a Orientação a Objetos consiste em considerar os sistemas computacionais não como uma coleção estruturada de processos, mas sim como uma coleção de objetos que interagem entre si [Farinelli 2007].

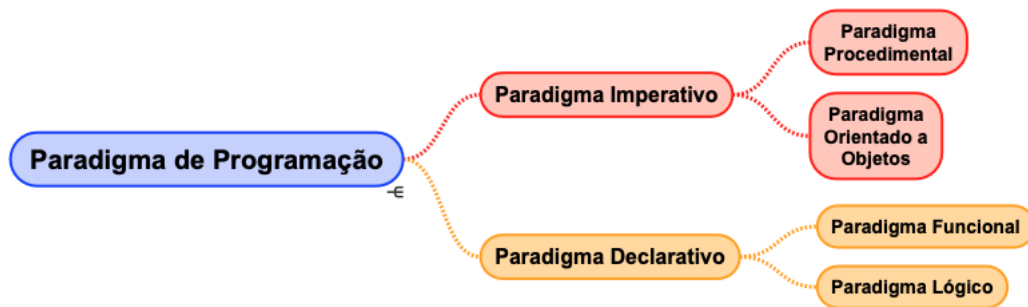
Figura 1 – Um sistema constituído de objetos que interagem entre si.



Fonte: [Sommerville 2003]

Segundo Peter Van Roy, um paradigma de programação define como a programação é realizada. Cada paradigma possui seu próprio conjunto de técnicas e formas de estruturar o pensamento na composição de software [Roy 2012]. Cada paradigma tem suas próprias vantagens e desvantagens e o desenvolvedor deve escolher o mais adequado para cada projeto e problema a ser resolvido.

Figura 2 – Paradigmas de Programação.



Fonte: O Autor

A programação orientada a objetos (POO) é um dos quatro principais paradigmas de programação, com o procedimental, funcional e lógico. A POO é uma abordagem para projetar e desenvolver software que se concentra na representação de entidades reais como objetos, com atributos e comportamentos. A programação orientada a objetos é realizada utilizando uma linguagem de programação orientada a objetos, como Java, que permite a implementação direta de objetos e fornece recursos para definir classes de objetos, como seus atributos e métodos. Esta abordagem permite a criação de programas de software mais organizados, reutilizáveis e fáceis de manter. A POO é amplamente utilizado em aplicações comerciais, games, sistemas web e muitos outros categorias de software, devido à sua capacidade de representar conceitos complexos de uma forma simples e fácil de entender. Além disso, a POO permite a reutilização de código, tornando o desenvolvimento de software mais rápido e eficiente [Sommerville 2003].

1.2 Principais conceitos de POO

A Programação Orientada a Objetos (POO), como já visto, é um paradigma de programação que se concentra em modelar conceitos reais como objetos computacionais, dotados de características e comportamentos. Esta abordagem permite a representação de dados e ações de forma mais intuitiva e natural, proporcionando uma melhor organização e reutilização de código em projetos de software. Abaixo será apresentado alguns dos principais conceitos envolvidos no paradigma.

1.2.1 Classes e objeto

Para [Sommerville 2003] “objeto” e “orientado a objetos” são amplamente utilizados e aplicados a diferentes categorias de entidades, métodos de projeto, sistemas e linguagens de programação. Contudo, existe uma aceitação geral de que um objeto é um encapsulamento de informações, e isso se reflete na definição de um objeto e de uma classe de objeto a seguir:

- Um objeto é uma entidade que possui um estado e um conjunto definido de operações que operam nesse estado. O estado é representado por um conjunto de atributos de objeto. As operações associadas com o objeto fornecem serviços para outros objetos (clientes), que requisitam esses serviços quando alguma computação é necessária.
- Os objetos são criados de acordo com uma definição de classe de objetos que serve como um template para criar objetos. Essa classe apresenta declarações de todos os atributos e operações que devem ser associados a um objeto dessa classe.

Em outras palavras, pode-se dizer que classe é uma descrição generalizada que descreve uma coleção de objetos similares, o qual, segundo Pressman e Maxim 2016, é um conceito orientado a objeto que encapsula dados e abstrações procedurais necessárias para descrever o conteúdo e comportamento de alguma entidade real.

Exemplos de objetos são: os **objetos físicos** (um livro, uma caneta), **funções de pessoas** para os sistemas (funcionário, cliente), **eventos** (uma compra, um telefonema), **interações** entre outros objetos (um item de uma nota fiscal é uma interação entre uma compra e um produto do estoque) e **lugares** (loja matriz, revenda nordeste).

Para fins de estudo, e com objetivo mais didático, usaremos um cachorro como nosso “objeto”:

Figura 3 – Representação de um objeto

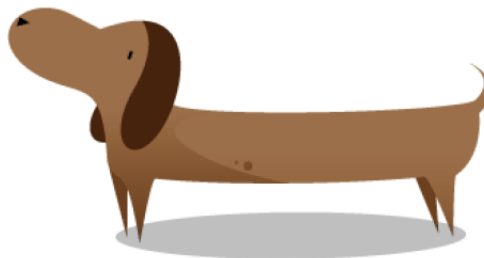


Figura 4 – Fonte: O autor

Ao analisar o objeto, deduz-se que há características pertencentes somente a ele. Tais como:

- Nome;
- Idade;
- Comprimento de pelos;
- Cor dos pelos;
- Cor dos olhos;
- Peso, entre outros;

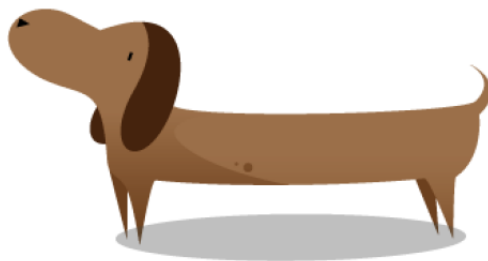
Tais características que descrevem um objeto são chamadas na orientação a objeto de atributos.

1.2.2 Atributos

Os objetos reais têm propriedades que, no que lhe concerne, possuem valores. Estes valores determinam o **estado do objeto**. Assim, na orientação a objeto, essas propriedades são chamadas atributos. Logo, podemos dizer que esses atributos são como variáveis ou campos que guardam os variados valores que os objetos podem receber como características.

O estado de um objeto é um grupo de valores que estão em seus atributos em um certo momento.

Figura 5 – Representação de um objeto



Fonte: O autor

Tabela 1 – Atributos e valores

Cachorro	
Nome:	Hubert
Idade:	2 anos
Tipo Pelo:	Curtos
Cor dos pelos:	Marrom
Cor dos olhos:	Castanhos
Peso	5kg

Fonte: O autor

Outro objeto cachorro teria valores diferentes para estes mesmos atributos, como exemplo disto temos:

Figura 6 – Representação de um objeto



Fonte: O autor

Tabela 2 – Atributos e valores

Cachorro	
Nome:	Floks
Idade:	4 anos
Tipo pelo:	Curtos
Cor dos pelos:	Branco
Cor dos olhos:	Castanhos
Peso	5kg

Fonte: O autor

Para que os atributos de um objeto mudem de valor isso deve ser feito exclusivamente por estímulos externos ou internos. Assim, a única maneira de alterar os atributos dos objetos é disparando eventos que geram a mudança desses estados no objeto.

1.2.3 Métodos

Os métodos são uma parte fundamental dos objetos em programação. Eles são procedimentos ou funções que executam ações específicas e permitem que o objeto se manifeste e interaja com outros objetos. Esses métodos são acionados por mensagens enviadas por outros objetos, que solicitam a realização de uma ação específica. Os métodos também são responsáveis por acessar e modificar os atributos do objeto.

Como exemplo, no estudo de um objeto "cachorro", podemos identificar uma série de métodos que ele possui, como latir, babar, comer e sentar. Esses métodos são o comportamento que o objeto cachorro pode exibir quando estimulado por outro objeto. Em resumo, os métodos são as ações que o objeto consegue realizar, e é através deles que ele se manifesta e interage no mundo virtual.

Já em sistemas computacionais, os métodos são funções ou procedimentos definidos em classes, ou objetos e executam tarefas específicas. Eles são utilizados para encapsular lógica de negócios, processamento de dados e outras operações que um objeto precisa realizar. Em resumo, eles são uma parte fundamental da programação orientada a objetos e são amplamente utilizados em sistemas computacionais para ajudar a organizar e modularizar o código, além de permitir a interação e reutilização de objetos.

1.2.4 Herança

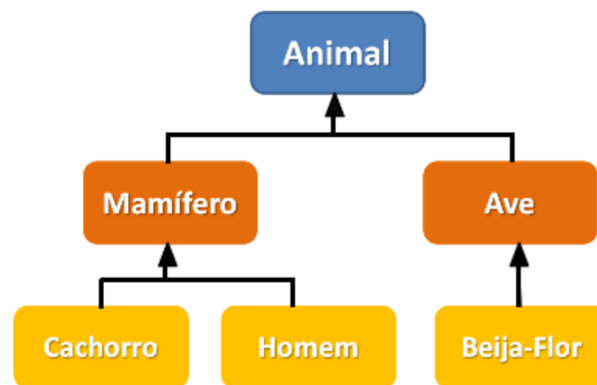
O conceito de herança é um dos conceitos fundamentais de POO. Herança, na prática, significa a possibilidade de construir objetos especializados que herdaram as características de objetos mais generalistas, ou ainda, a herança uma maneira de reutilizar código a medida que podemos aproveitar os atributos e métodos de classes já existentes para

gerar novas classes mais específicas que aproveitarão os recursos da classe hierarquicamente superior [Ruiz 2008].

O conceito de herança mimetiza as características hierárquicas de vários sistemas reais, como, por exemplo, os sistemas de classificação em biologia que, pode determinar como uma hierarquia o seguinte:

- animais;
- vertebrados e invertebrados;
- mamíferos e aves;
- entre outras características mais específicas

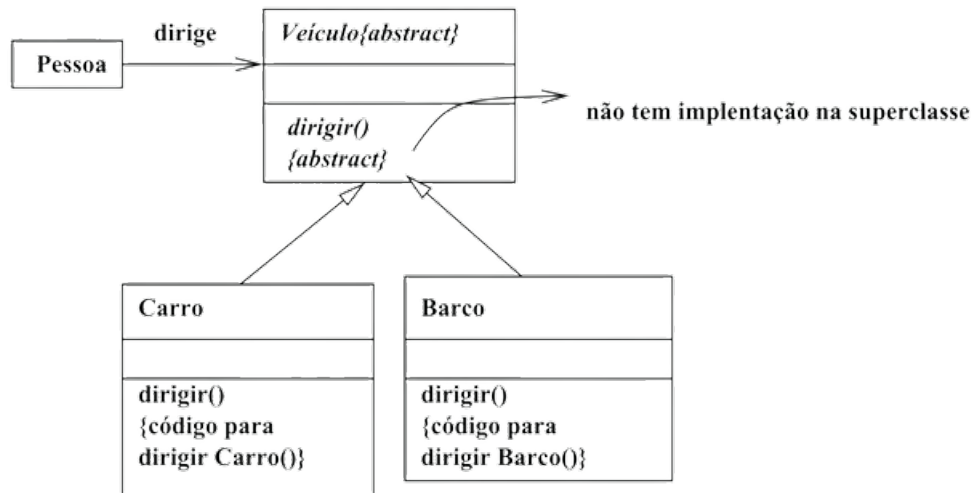
Figura 7 – Diagrama representando a herança entre as classes



Fonte: ...

1.3 Classes abstratas

As classes abstratas são moldes para outras classes que herdam seus atributos e métodos. Elas não podem ser instanciadas diretamente, mas precisam ser estendidas por uma classe mais específica, a qual pode ser instanciada. Os métodos da classe abstrata precisam ser redefinidos nas classes filhas (subclasses), permitindo a personalização do comportamento do objeto.

Figura 8 – Diagrama representando uma classe abstrata

Fonte: <<https://pt.slideshare.net/CristianoSilva11/class-abstrata-java>>

1.3.1 Superclasses e subclasses

Em POO todo objeto de uma classe construída pelo usuário da linguagem é também um objeto de outra classe. Por exemplo, na hierarquia de uma empresa, podemos dizer que pessoa é uma superclasse e que funcionário é uma subclasse de pessoa.

Outra nomenclatura utilizada para especificar superclasses ou subclasses é a Generalização, ou Especialização. No exemplo abaixo, pessoa é a generalização de empregado, e empregado é a especialização de pessoa conforme representado na Figura 9

Uma máxima que podemos guardar é:

Uma subclasse guarda a relação é um com a superclasse.

Figura 9 – Superclasses e Subclasses

Fonte: [Silva et al. 2009]

Exercícios de fixação

1. Para satisfazer as necessidades de informatização de uma biblioteca universitária um sistema foi proposto para satisfazer algumas características:
 - Cadastro dos usuários da biblioteca com endereço completo. Usuário são classificados em três grupos: professores, alunos e funcionário.
 - Cadastro das obras da biblioteca são classificados em: livros científicos, periódicos científicos, periódicos informativos, periódicos diversos, entretenimento, etc.
 - Linguagem usada no exemplar da obra.
 - Mídia que armazena o exemplar da obra.
 - Autores da obra com o controle da nacionalidade dos mesmos.
 - Editoras dos exemplares com ano de edição referente a cada exemplar.

Identifique os possíveis objetos com seus respectivos atributos e métodos.

2. **Desafio - Obrigatório:** Pesquise sobre os pontos negativos da orientação a objeto, principalmente sobre os conceitos que relacionam **Coesão** e **Acoplamento**.

1.4 Mensagem

Mensagens são requisições enviadas de um objeto para outro, para que o objeto “receptor” forneça algum resultado desejado por meio da execução de uma operação. As

trocas de mensagem funcionam como uma fábrica que recebe uma ordem de produção (mensagem de solicitação), processa essa ordem (operações) utilizando matéria-prima (atributos) e gera um produto final (mensagem de resposta).

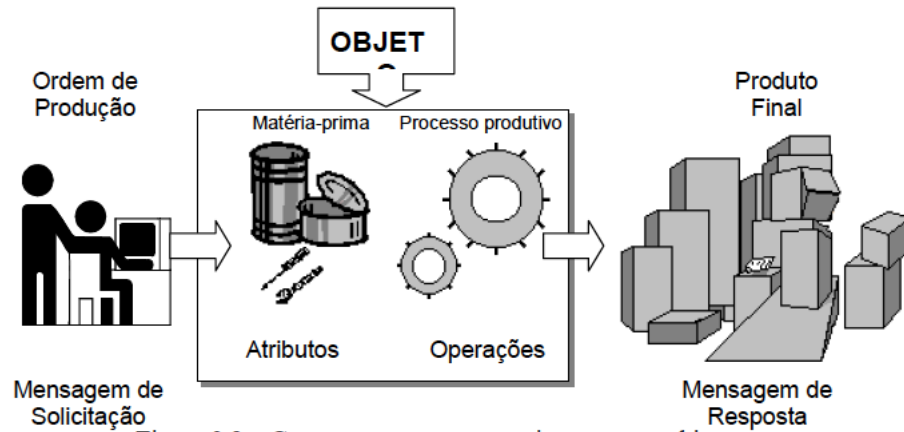


Figura 10 – Comportamento e comunicação entre objetos

1.5 Encapsulamento

Cada objeto é visto como o encapsulamento do seu estado interno, suas mensagens e seus métodos. A estrutura do estado e dos pares "mensagem - método" são todas definidas através da classe à qual o objeto pertence.

- A encapsulação de dados com o código que os manipula em classes é a principal vantagem da Orientação a Objeto
- No sentido de não quebrar a encapsulação, é muito importante que os membros de uma classe (atributos e métodos) sejam visíveis apenas onde estritamente necessário (A lei é: "Não posso quebrar o que não posso acessar").

1.5.1 Especificadores de controle de acesso

Com a ideia de encapsulamento nós também possuímos os especificadores de controle de acesso ou visibilidades. A visibilidade é a maneira com a qual o desenvolvedor proíbe ou permite acesso a determinados métodos, ou atributos de uma classe, como pode ser visto na Figura 11. Neste sentido, o objeto formado possuirá as mesmas definições declaradas pela classe.

1.5.1.1 A visibilidade *public*

- Quem tem acesso à classe tem acesso também a qualquer membro com visibilidade *public*;

- O alvo aqui é o programador cliente que usa suas classes;
- É raro ter atributos públicos, mas é comum ter métodos públicos.

1.5.1.2 A visibilidade *private*

- O membro *private* não é acessível fora da classe;
- A intenção aqui é permitir que apenas você que escreve a classe possa usar esse membro.

1.5.1.3 A visibilidade *protected*

- O membro *protected* é acessível à classe e a suas subclasses;
- A intenção é dar acesso aos programadores que estenderão sua classe.

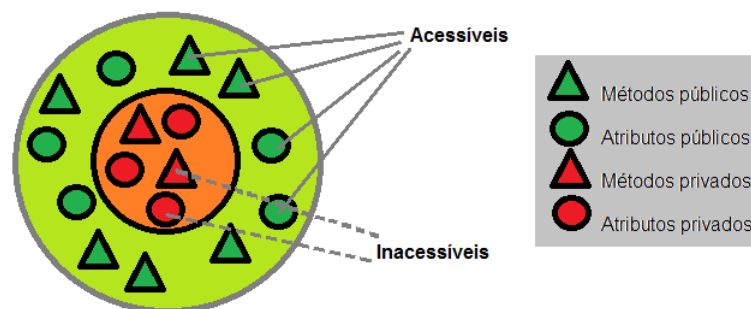


Figura 11 – Encapsulamento

1.6 Polimorfismo

É a propriedade que permite que a mesma mensagem seja enviada a diferentes objetos e que cada objeto execute a operação apropriada à sua classe. No caso de polimorfismo, é necessário que os métodos tenham a mesma identificação, utilizando o mecanismo de redefinição de métodos.

No exemplo utilizado na Figura 12, podemos perceber que diferentes objetos, quando é solicitada a mesma ação, se comportam de maneira diferente. Similar a objetos reais, na Orientação a Objetos, o comportamento por meio do polimorfismo acontece da mesma forma.

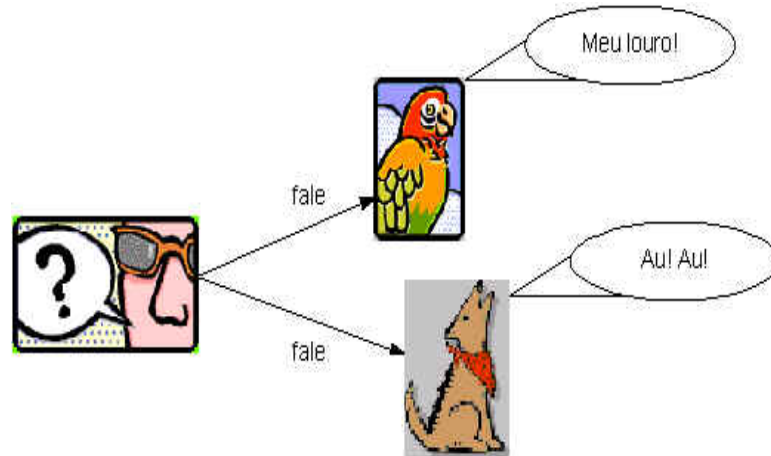


Figura 12 – Polimorfismo

1.7 Conceitos estudados até este momento

Para ilustrar, a tabela abaixo contém um resumo e um exemplo de estes conceitos.

Palavra-Chave	Breve Definição	Exemplo
Classe	Agrupamento de objetos similares que apresentam os mesmos atributos e operações	Indivíduo, caracterizando as pessoas do mundo
Atributo	Característica particular de uma ocorrência da classe	Indivíduo possui nome, sexo, data de nascimento
Operações	Lógica contida em uma classe para designar-lhe um comportamento	Cálculo da idade de uma pessoa em uma classe (Indivíduo)
Encapsulamento	Combinação de atributos e operações de uma classe	Atributo: data de nascimento Operação: cálculo da idade
Herança	Compartilhamento pela subclasse dos atributos e operações da classe pai	Subclasse (Eucalipto) compartilha atributos e operações da classe (Árvore)
Subclasse	Característica particular de uma classe	Classe (Árvore), Subclasses (Ipê, Eucalipto, Jacarandá, etc.)
Instância de Classe	Uma ocorrência específica de uma classe. É o mesmo que objeto	Uma pessoa, uma organização ou um equipamento
Objeto	Elemento do mundo real (natureza). Sinônimo de instância de classe	Pessoa “Fulano de Tal”, Organização “ACM”, Equipamento “Extintor”
Mensagem	Uma solicitação entre objetos para invocar certa operação	Informar idade da pessoa “Fulano de Tal”
Polimorfismo	Habilidade para usar a mesma mensagem para invocar comportamentos diferentes do objeto	Chamada da operação: “Calcular Saldo” de correntista. Invoca as derivações correspondentes para cálculo de saldo de poupança, renda fixa, etc.

Exercícios de fixação

Baseando-se nas explicações em aula e em suas anotações, responda as seguintes questões:

1. Um ponto importante que deve ser claro para podermos atingir o objetivo de nossa disciplina é o conceito sobre classes e objetos. Explique o que são e de exemplos do seu cotidiano para reforçar a ideia.

2. O que são atributos e métodos? Defina cada um e de exemplos para afirmar sua resposta.
3. Um dos pilares da orientação a objeto é a Herança. Desenhe um gráfico, semelhante ao da Figura 7, representando a sua Árvore Genealógica. Caso consiga, vá até seus bisavós.
4. Utilizando a atividade anterior. Identifique os possíveis objetos que podem ser herdados de outros objetos. Utilize setas que apontam do objeto **Especializado** para o **Generalizado**.
5. Utilizando a atividade anterior. Represente, por setas, as possíveis trocas de mensagem entre os objetos que você pode identificar.

2 Orientação a Objetos com JavaScript

*Dedique-se aos estudos para que eles o façam
melhor para a sociedade e para si mesmo.*

Alvaro Granha Loregian

Esse capítulo tem como objetivo aplicar o conhecimento adquirido anteriormente no Capítulo 1 utilizando a linguagem de programação JavaScript.

JavaScript, segundo Trasviña 2021, tem fortes capacidades de programação orientada a objetos, apesar de ocorrerem algumas discussões devido às diferenças da orientação a objetos no JavaScript em comparação com outras linguagens. Assim, no Capítulo 1 foi realizada uma introdução à programação orientada a objetos, e neste, será demonstrado os conceitos de programação orientada a objetos no JavaScript.

2.1 Classes

Classes em JavaScript¹ são introduzidas no ECMAScript² 2015 e são simplificações da linguagem para as heranças baseadas nos protótipos. A sintaxe para classes não introduz um novo modelo de herança de orientação a objetos em JavaScript. Classes em JavaScript provêm uma maneira mais simples e clara de criar objetos e lidar com herança.



O diagrama ilustra a transição da sintaxe de objetos literais para a sintaxe de classes. À esquerda, um objeto literal `Usuario` é definido com propriedades `nome` e `profissao`, e é acessado via `console.log(Usuario.nome)`. Uma seta aponta para a direita, onde a mesma lógica é mostrada usando a sintaxe de classes: `class Usuario` com um construtor que atribui `this.nome` e `this.profissao`, seguido pela criação de uma instância `usuario = new Usuario()` e seu log.

```
let Usuario = {  
  nome: 'Luiz Picolo',  
  profissao: 'Professor'  
}  
  
console.log(Usuario.nome)
```

```
class Usuario {  
  constructor(){  
    this.nome = 'Luiz Picolo';  
    this.profissao =  
    'Professor'  
  }  
}  
  
usuario = new Usuario();  
console.log(usuario.nome)
```

Figura 13 – Sintaxes de classes em JavaScript

Fonte: O Autor

Para definir uma classe é usando uma declaração de classe. Para declarar uma classe, deve-se usar a palavra reservada **class** seguida pelo nome da classe desejada.

¹ Para mais detalhes acesse: <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Classes>>

² Para mais detalhes acesse: <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>>

```
1  class Retangulo {  
2  
3  }
```

Outras forma de escrita de uma classe, também chamadas Expressões de Classes (*class expression*), podem ser usadas para fazer sua definição. Contudo, utilizaremos apenas a notação mais nova e comumente encontrada em outras linguagens, ou seja, a notação *class*.

```
1  // sem nome  
2  let Retangulo = class {  
3  
4  };  
5  
6  // nomeada  
7  let Retangulo = class Retangulo {  
8  
9  };
```

E para que possamos criar uma instância (objeto) da classe, utilizamos o seguinte trecho de código abaixo.

```
1  let retangulo = new Retangulo();
```

2.2 Construtor e Atributos

Como dito no Capítulo 1 as características que descrevem um objeto são chamadas na orientação a objeto de atributos. Assim, toda classe pode conter atributos que lhe dão as características necessárias para a criação do objeto.

Para que possamos criar nossos atributos podemos fazer de duas formas, uma usando funções diretas ou por meio do Construtor. Um construtor é um tipo especial de “método” para criar e iniciar um objeto criado pela classe. Só pode existir um método construtor dentro da classe. Um erro de sintaxe **SyntaxError** será lançado se a classe possui mais do que uma ocorrência do método construtor.

Para iniciarmos os atributos com o construtor usamos a palavra reservada **constructor** seguida dos valores dos atributos desejados.

```
1  class Retangulo {  
2      constructor(altura, largura) {  
3          this._altura = altura;  
4          this._largura = largura;  
5      }  
6  }
```

2.3 Métodos

Métodos são ações que o objeto pode realizar e são responsáveis pela troca de mensagem entre os objetos **emissores** e o **receptores**. As técnicas de programação orientada a objetos recomendam que a estrutura de um objeto e a implementação de seus métodos devem ser tão privativos como possível. Normalmente, os atributos de um objeto não devem ser visíveis externamente. Da mesma forma, de um método deve ser suficiente conhecer apenas sua especificação, sem necessidade de saber detalhes de como a funcionalidade que ele executa é implementada³.



Figura 14 – Troca de mensagens entre objetos.

Fonte: O Autor

Assim, para definir um método, devemos atribuir uma função a uma classe que, depois disso, poderá ser chamado como método do objeto usando o mesmo nome ao qual foi atribuída a função.

```
1  class Retangulo {  
2      constructor(altura, largura) {
```

³ Veja mais em: <<https://www.devmedia.com.br/orientacao-a-objetos-parte-ii/7161>>


```
3     this._altura = altura;
4     this._largura = largura;
5 }
6
7 calcularArea() {
8     return this._altura * this._largura;
9 }
10 }
```

Para invocarmos o método, basta instanciar-mos a classe e criar assim um novo objeto.

```
1 const objeto = new Retangulo(10, 10);
2 console.log(objeto.calculaArea());
```

Contudo, existe aqui um erro semântico. Quando queremos saber a área de uma figura geométrica, deveríamos “perguntar” a está figura sua área e não solicitar o calculo. Logo, podemos resolver isso, criando um novo método que retorna apenas a área e que utiliza o método **calcularArea**.

```
1 class Retangulo {
2     constructor(altura, largura) {
3         this._altura = altura;
4         this._largura = largura;
5     }
6
7     area(){
8         return this.calcularArea();
9     }
10
11     calcularArea() {
12         return this._altura * this._largura;
13     }
14 }
15
16 const objeto = new Retangulo(10, 10);
17 console.log(objeto.area());
```

2.3.1 Método Get

Os métodos criados acima funcionam corretamente. Porém, podemos melhorar um pouco mais a semântica do objeto. Para isso usaremos a sintaxe **Get**.

Às vezes é desejável que se permita acesso a uma propriedade que retorna um valor computado dinamicamente, ou você pode querer refletir o status de uma variável interna sem requerer o uso de chamadas de método explícitas. Em Javascript, isso pode ser feito com o uso de um **getter**. Não é possível simultaneamente ter um **getter** associado a uma propriedade e a mesma possuir um valor, embora seja possível usar um **getter** e um **setter** em conjunto para criar algo como uma pseudo-propriedade.

```
1  class Retangulo {
2      constructor(altura, largura) {
3          this._altura = altura;
4          this._largura = largura;
5      }
6
7      get area(){
8          return this.calcularArea();
9      }
10
11     calcularArea() {
12         return this._altura * this._largura;
13     }
14 }
15
16 const objeto = new Retangulo(10, 10);
17 console.log(objeto.area);
```

2.3.2 Método Set

Já o **setter** pode ser usado para executar uma função sempre que se tenta mudar uma propriedade específica. **Setters** são geralmente usados em conjunto com **getters**, para criar um tipo de pseudo-propriedade. No entanto é impossível ter-se um **setter** para uma propriedade que contenha um valor real.

```
1  class Retangulo {
2
```

```
3     set altura(altura){
4         this._altura = altura;
5     }
6
7     set largura(largura){
8         this._largura = largura;
9     }
10
11    get area(){
12        return this.calcularArea();
13    }
14
15    calcularArea() {
16        return this._altura * this._largura;
17    }
18 }
19
20 const objeto = new Retangulo();
21 objeto.largura = 10;
22 objeto.altura = 10;
23 console.log(objeto.area);
24
25 objeto.largura = 100;
26 objeto.altura = 100;
27 console.log(objeto.area);
```

O setter se torna muito útil nesse caso pois não precisaremos criar um novo objeto, apenas alteramos seus atributos (seu estado) para obter novos valores.

2.3.3 Método Static

A palavra chave **static** define um método estático para a classe. Métodos estáticos não são chamados nas instâncias da classe. Em vez disso, eles são chamados na própria classe. Geralmente, são funções utilitárias, como funções para criar ou clonar objetos. Chamadas a métodos estáticos são feitas diretamente na classe e não podem ser feitas em uma instância da classe. Métodos estáticos são comumente utilizados como funções utilitárias⁴.

⁴ Veja mais em: <<https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference/Classes/static>>

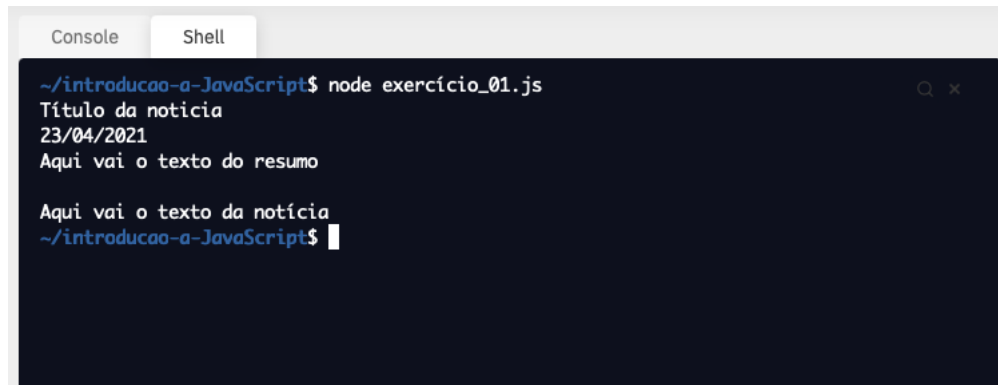
```
1  class Retangulo {
2
3      //Codigo anterior
4
5      //Metodo adicionado
6      static formaGeometrica(objeto){
7          if (objeto._altura == objeto._largura){
8              return `A forma geometrica é um Quadrado`
9          } else {
10             return `A forma geometrica é um Retangulo`
11          }
12      }
13  }
14
15  const objeto = new Retangulo();
16  objeto.largura = 10;
17  objeto.altura = 10;
18  console.log(Retangulo.formaGeometrica(objeto));
19  console.log(objeto.area);
```

2.4 Exercícios de fixação

1. Uma empresa de notícias **IFNews** necessita desenvolver um portal de notícias. Após criada toda a parte documental, você, que é desenvolvedor/desenvolvedora na empresa, deverá iniciar a criação dos primeiros códigos. Sua primeira tarefa é a criação da classe **Noticia**. A classe deverá ser codificada seguindo, primordialmente, as definições abaixo:
 - a) **Nome da classe:** **Noticia**
 - b) **Atributos:** Título, Data da Publicação, Resumo e Texto (Outros atributos podem ser adicionados)
 - c) **Métodos:** Mostrar Noticia (Outros métodos podem ser adicionados)

Ao instanciar a classe e invocar o método **mostrar notícia** o resultado final deverá ser semelhante ao descrito abaixo (Lembre-se que o conteúdo no título, a data e o texto dependerá do **estado** adicionado a seu objeto):

Resultado esperado:



```
~/introducao-a-JavaScript$ node exercicio_01.js
Título da noticia
23/04/2021
Aqui vai o texto do resumo

Aqui vai o texto da notícia
~/introducao-a-JavaScript$
```

Figura 15 – Resultado esperado

Fonte: O Autor

2.5 Exercícios extras

1. Crie uma classe para representar uma **pessoa**, com os atributos **nome**, **data de nascimento** e **altura**. Crie o método **constructor** para inicializar os atributos e também um método para imprimir todos dados de uma pessoa e outro para calcular a idade.
2. Crie uma classe para representar um **jogador de futebol**, com os atributos **nome**, **posição**, **data de nascimento**, **nacionalidade**, **altura** e **peso**. Crie o método para imprimir todos os dados do jogador. Crie um método para calcular a idade do jogador e outro método para mostrar quanto tempo falta para o jogador se aposentar. Para isso, considere que os jogadores da posição de defesa se aposentam em média aos 40 anos, os jogadores de meio-campo aos 38 e os atacantes aos 35.
3. Escreva uma classe cujos objetos representam alunos matriculados em uma disciplina. Cada objeto dessa classe deve guardar os seguintes dados do aluno: matrícula, nome, 2 notas de prova e 1 nota de trabalho. Escreva os seguintes métodos para esta classe:
 - **media**: calcula a média final do aluno (cada prova tem peso 2,5 e o trabalho tem peso 2)
 - **final**: calcula quanto o aluno precisa para a prova final (retorna zero se ele não for para a final)
4. Crie uma classe Agenda que pode armazenar 10 pessoas e que seja capaz de realizar as seguintes operações:

```
armazena(nome, idade, altura);
remove(nome);
```

```
busca(nome); // informa em que posição da agenda está a pessoa
imprimeDadosAgenda(); // imprime os dados de todas as pessoas da agenda
mostrar(index); // imprime os dados da pessoa que está na posição “i” da agenda.
```

2.6 Herança

A definição de herança, já abordada no Capítulo anterior, nós diz que: a herança é uma maneira de reutilizar código a medida que podemos aproveitar os atributos e métodos de classes já existentes para gerar novas classes mais específicas que aproveitarão os recursos da classe hierarquicamente superior [Ruiz 2008].

Para realizar a reutilização já citada, no ES6 foi adicionada a palavra reservada *extends*. Uma classe pode herdar atributos e métodos de outra classe por meio da extensão da classe desejada.

```
1  class Retangulo {
2
3      // Codigo anterior
4      ...
5  }
6
7  class Quadrado extends Retangulo{
8      constructor(lado) {
9          super(lado, lado);
10     }
11 }
12
13 let objeto = new Quadrado(100)
14 console.log(objeto.area)
```

Observem a palavra **super**. A palavra reservada **super** é usada para acessar o objeto pai de um objeto, em outros casos, é usada para acessar a classe pai de uma classe. Neste caso, precisamos observar alguns pontos:

- No construtor, o **super** deve ser chamado antes de qualquer acesso à **this**;
- Caso a classe que está realizando *extends* não defina o constructor ele será definido e chamado implicitamente passando todos os parâmetros.

```
1 // Errado
2 class Quadrado extends Retangulo{
3     constructor(lado) {
4         this._lado = lado;
5         super(lado, lado);
6     }
7 }
8
9 // Correto
10 class Quadrado extends Retangulo{
11     constructor(lado) {
12         super(lado, lado);
13         this._lado = lado;
14     }
15 }
```

2.7 Super classe e Subclasse

Arelado com o conceito de herança possuímos as definições de **Super classe** e **Subclasses**. O uso da palavra **extends** indica que a classe deveria ser baseado em um objeto padrão. Isso é chamado de Super Classe (*superclass*) e a classe derivada é a Subclasse (*subclass*) [Haverbeke 2018]

```
1 class Retangulo {
2     ...
3 }
4
5 class Quadrado extends Retangulo{
6     ...
7 }
```

Ocasionalmente, é útil saber se um objeto foi derivado de uma classe. Para isso, o JavaScript fornece um operador binário chamado **instanceof** [Haverbeke 2018].

```
1 console.log(new Quadrado(2) instanceof Retangulo) // True
```

Mas e **Retangulo** é instância de alguma classe? Neste caso sim, todos os objetos em JavaScript herdam a classe pai nomeado de **Object**

```
1 console.log(new Quadrado(2) instanceof Object) \\ True
2 console.log(new Retangulo(2, 2) instanceof Object) \\ True
```

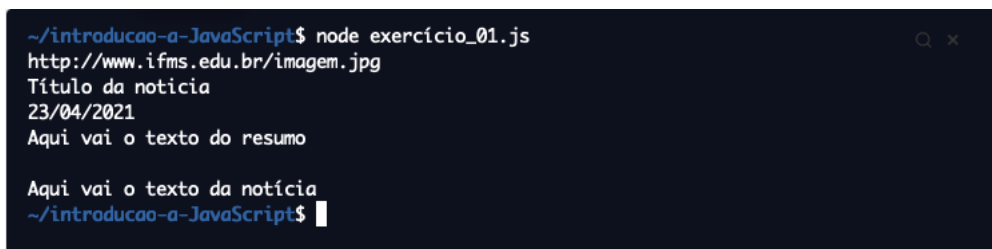
2.8 Exercícios de fixação

1. A empresa de notícias **IFNews** contratou você para desenvolver o portal de notícias, o qual teve seu início no exercício anterior. Sua primeira tarefa foi a criação da classe **Noticia** que possuía as seguintes definições:
 - a) **Nome da classe:** **Noticia**
 - b) **Atributos:** Título, Data da Publicação, Resumo e Texto (Outros atributos podem ser adicionados)
 - c) **Métodos:** Mostrar Noticia (Outros métodos podem ser adicionados)

Após desenvolver a tarefa, a empresa criou uma nova solicitação. O portal deve possuir **notícias em destaque**, as quais devem conter uma imagem seguida de um título e um resumo. Como você já criou uma notícia que contem parte do que o cliente necessita, a única tarefa será criar uma **subclasse** que herdará as características e ações da **Super Classe** e adicionará a característica desejada.

1. **Nome da classe:** **NoticiaDestaque**
2. **Atributos:** ImagemDestaque (apenas o caminho da imagem)
3. **Métodos:** Mostrar Destaque (Outros métodos podem ser adicionados)

Resultado esperado:



```
~/introducao-a-JavaScript$ node exercicio_01.js
http://www.ifms.edu.br/imagem.jpg
Título da noticia
23/04/2021
Aqui vai o texto do resumo

Aqui vai o texto da notícia
~/introducao-a-JavaScript$
```

Figura 16 – Resultado esperado

Fonte: O Autor

2.9 Exercícios extras

1. Crie classes de forma a representar o diagrama a abaixo:



2.10 Tratamento de erros

Um requisito de software que deveríamos priorizar, em qualquer projeto, é como vamos realizar o tratamento de erros e exceções da nossa aplicação. **Saber como gerenciar erros é parte importante da programação.**

Existem dois tipos básicos de erros na programação:

- Erros de sintaxe;
- Erros de tempo de execução;

2.10.1 Erros de sintaxe

Os erros de sintaxe, também conhecidos como erros de análise, no tempo de interpretação no *JavaScript* (Linguagem interpretada / Linguagem compilada). Esse erro ocorre porque ao digitarmos algum código errado, ou seja, no momento em que esquecemos parênteses ou chaves, etc.

```
1 class Retangulo {
2   constructor(altura, largura){
3     this._altura = altura;
4     this._largura = largura;
5   }
6   // Falta fechar as chaves
```

Ao executar o código acima, o retorno será: **SyntaxError: Unexpected end of input**

Os erros de Sintaxe são erros mais simples de serem resolvidos. Existem algumas maneiras de evitar esses tipos de erros, as quais são:

- Verificar o seu código e procurá-los (não é eficaz e pode levar muito tempo)
- Outra opção é usar alguma ferramenta ou plugin IDE. (eslint , jshint e jslint)

Essas ferramentas são muito populares e os(as) desenvolvedores(as), não gostam de tarefas repetitivas.

2.10.2 Erros de tempo de execução

Erros de execução, também chamados de **exceções**, ocorrem durante a execução (após a compilação/interpretação). Este tipo de erro não impede a compilação ou interpretação de um código. Contudo, o código gerado não funcionará.

```
1  class Retangulo {  
2      constructor(altura, largura){  
3          this._altura = altura;  
4          this._largura = largura;  
5      }  
6  }  
7  
8  let retangulo = new Retangulos(); // Nome no plural
```

Ao executar o código acima, o retorno será: *ReferenceError: Retangulos is not defined*

Por esse ser um erro mais complexo de ser tratado algumas linguagens implementam alguns mecanismos interessantes para manipular o retorno dos erros, sendo que, as exceções são utilizadas como uma forma de *feedback*, para que o(a) desenvolvedor(ar) possa saber o que fazer quando cair numa situação de erro.

Assim, para capturar estes erros e aplicar algum tratamento, podemos usar a sintaxe **try catch** a qual será o assunto a seguir.

2.10.3 Sintaxe do Try Catch Finally

A sintaxe do **try catch** é composta por dois blocos principais:

- O *try* é o local na qual o código é executado
- O *catch*, e o local que você recebe, via parâmetro na função, um **objeto do tipo Error**.

```
1  try {  
2    // seu codigo aqui  
3  } catch (error) {  
4    // tratamento de erro aqui  
5  }
```

2.10.4 Sintaxe do Try / Catch / finally

Nós podemos também usar a cláusula **finally**. Este bloco será executado sempre independente se houver ou não falha. Depois que o *try* ou *catch* serem executados, este bloco será acionado.

Isto pode ser útil por exemplo, para fechar um arquivo que foi aberto para leitura, registrar algum log ou fechar alguma conexão. Veja a sintaxe de exemplo.

```
1  try {  
2    // seu codigo aqui  
3  } catch (error) {  
4    // tratamento de erro aqui  
5  } finally {  
6    // executa sempre  
7  }
```

2.11 Exercícios de fixação

1. A classe Retangulo não possui tratamento de erros. Caso o usuário aplique uma altura e/ou uma largura igual ou menor que **Zero** ele não receberá um erro para que possa corrigir o problema. Assim, utilizando o conceito apresentado anteriormente, aplique o **Try** e o **Catch** no código abaixo para que, caso o usuário atribua um valor menor ou igual a zero, o código retorne um erro apontando como solucioná-lo.

```
1  class Retangulo {  
2    constructor(altura, largura){  
3      this._altura = altura;  
4      this._largura = largura;  
5    }
```

```
6
7   get area(){
8       return this.calculaArea(this._altura, this._largura)
9   }
10
11   calculaArea(altura, largura){
12       return altura * largura
13   }
14 }
```

2.11.1 Objeto Error

Sempre que uma exceção é lançada dentro do bloco `try`, o *JavaScript* cria um **objeto** do tipo **Error** e envia como argumento para o `catch`. Por padrão, este objeto é composto de duas propriedades principais:

- **name**: Representa o tipo do erro. Por exemplo, um erro de sintaxe - `SyntaxError`.
- **message**: Mensagem em texto, contendo mais detalhes do erro.

Em alguns ambientes onde você realiza o tratamento de erros em *JavaScript*, também terá a propriedade **stack**. Nela você tem a sequência de chamadas que levaram ao erro. Este tipo de informação é bastante útil para depuração.

2.11.2 Tipos nativos de erros

O objeto **Error** possui alguns tipos nativos que são invocados dependendo do contexto do erro. O JavaScript possui os seguintes tipos de erros nativos que podem ser lançados:

ReferenceError Lançado quando uma referência a uma variável ou função inexistente ou inválida é detectada.

TypeError Lançado quando um operador ou argumento passado para a função é de um tipo diferente do esperado.

SyntaxError Lançado quando ocorre algum erro de sintaxe ao interpretar o código, por exemplo ao realizar o parse de um JSON.

URIError Lançado quando ocorre algum erro no tratamento de URI, por exemplo, enviando parâmetros inválidos no `decodeURI()` ou `encodeURI()`.

RangeError Lançado quando um valor não está no conjunto ou intervalo de valores permitidos. Por exemplo, um valor em string num array número.

2.11.3 Customizar erros

Os erros nativos do JavaScript são muito úteis pelo fato de não termos ideia do problema que pode ocorrer. Contudo, se possuírmos regras de negócio específicas, podemos criar nossos próprios tipos de erros e lançar quando for preciso.

Para tanto, devemos estender a classe `Error` e criar uma modificação em seu construtor.

```
1  class ErroCustomizado extends Error {  
2      constructor(message) {  
3          super(message)  
4          this.name = 'ErroCustomizado'  
5      }  
6  }  
7  
8  class Exemplo {  
9      metodo(){  
10         try {  
11  
12         } catch(e){  
13  
14         }  
15     }  
16 }
```

2.11.4 Operador Throw

No exemplo anterior nos utilizamos o operador **throw** para lançar o erro. Quando há a necessidade de lançar uma exceção, usamos o operador **throw** precedida do valor, e este pode ser:

- string
- número
- objeto literal
- função ou até mesmo uma classe

```
1  throw "Descricao do erro"  
2
```

```
3      throw 404
4
5      throw {
6          name: "Erro",
7          message: "Descricao do erro"
8      }
9
10     throw new UserTypeError("Tipo de usuario invalido")
```

Assim, para que possamos aplicar o erro customizado faríamos da seguinte forma:

```
1  class ErroCustomizado extends Error {
2      constructor(message) {
3          super(message)
4          this.name = 'ErroCustomizado'
5      }
6  }
7
8  class Exemplo {
9      metodo(){
10         try {
11             ...
12         } catch(e){
13             throw new ErroCustomizado("Mensagem de erro")
14         }
15     }
16 }
```

Um exemplo, mais completo com a aplicação de todo o conhecimento sobre Customização de Erros pode ser encontrado no link a seguir <<https://gist.github.com/luizpicolo/cad5da3bb233a34c9c7759423ab2da53>>

2.12 Exercícios de fixação

1. Com base na classe **Retangulo** do exercício anterior, crie e aplique o conceito de erro customizado.

2. Com base nas classe Notícias e **NoticiaDestaque** implementadas em exercícios anteriores, crie e aplique o conceito de erro customizado.

3 Definição sobre JavaScript Object Notation (JSON)

*Um ladrão rouba um tesouro, mas não furta a inteligência.
Uma crise destrói um herança, mas não uma profissão.
Não importa se você não tem dinheiro, você é uma pessoa rica,
pois possui o maior de todos os capitais: a sua inteligência.
Invista nela. **Estude!**
Augusto Cury*

JSON (JavaScript Object Notation) é um formato de intercâmbio de dados leve. É fácil para que seres humanos possam ler e escrever. É fácil para que as máquinas possam analisar e gerar. É baseado em um subconjunto do **Padrão de Linguagem de Programação JavaScript ECMA-262 3ª Edição** - Dezembro de 1999. JSON é um formato de texto completamente independente da linguagem, mas usa convenções que são familiares aos programadores da família C de linguagens, incluindo C, C ++, Java, JavaScript, Perl, Python e muitos outros. Essas propriedades tornam o JSON uma linguagem de intercâmbio de dados ideal¹.

Json também é um formato baseado em texto padrão para representar dados estruturados com base na sintaxe do objeto JavaScript. É comumente usado para transmitir dados em aplicativos da Web (por exemplo, enviar alguns dados do servidor para o cliente, para que possam ser exibidos em uma página da Web ou vice-versa). Você se deparará com isso com bastante frequência, portanto, neste artigo, oferecemos tudo o que você precisa para trabalhar com o JSON usando JavaScript, incluindo a análise do JSON para que você possa acessar os dados dentro dele e criar o JSON².

3.1 Estrutura JSON

Um JSON é uma string cujo formato se parece muito com o formato literal do objeto JavaScript. Você pode incluir os mesmos tipos de dados básicos dentro do JSON, como em um objeto JavaScript padrão — strings, números, matrizes, booleanos e outros literais de objeto. Isso permite que você construa uma hierarquia de dados.

¹ Para mais detalhes acesse: <https://www.json.org/json-pt.html>

² Para mais detalhes acesse: <<https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Objects/JSON>>


```
1 let Retangulo = {
2   "largura": 100,
3   "altura": 100
4 }
5
6 let Retangulos = [
7   {"largura": 100, "altura": 100},
8   {"largura": 200, "altura": 200},
9 ]
```

Assim, para realizar o acesso aos dados de um objeto Json, usamos a mesma notação **dot** / **bracket** usada em objetos literais.

```
1 let Retangulo = {
2   "largura": 100,
3   "altura": 100
4 }
5
6 let Retangulos = [
7   {"largura": 100, "altura": 100},
8   {"largura": 200, "altura": 200},
9 ]
10
11 console.log(Retangulo.largura)
12 console.log(Retangulo['largura'])
13
14 console.log(Retangulos[0].largura)
15 console.log(Retangulos[0]['largura'])
```

Contudo, o Json recebido venha em formato de String, ele deverá ser convertido para um objeto JavaScript. Assim, usando a função **JSON.parse()** podemos converter a string em um objeto, da seguinte forma:

```
1 let retangulo = Json.parse('{ "largura": 100, "altura": 100 }')
2 console.log(Retangulo.largura)
```

3.2 Exercícios de Fixação

1. No exercício 2.4 criamos uma classe **Noticia** que continha alguns atributos (titulo, dataDaPublicacao, resumo e texto) e uma outra classe **NoticiaDestaque** que continha, além dos atributos citados, um atributo imagem. Assim, baseando-se nos atributos listados, crie um arquivo **json** com nome **noticias.json** e atribua a ele uma lista de notícias no formato estudado. O arquivo deve conter no mínimo 5 notícias.

3.3 Obtendo JSON de um servidor web (Requisição Ajax)

Para obter o JSON, vamos usar uma API chamada **XMLHttpRequest** (geralmente chamada de XHR). Esse é um objeto JavaScript muito útil que nos permite fazer solicitações de rede para recuperar recursos de um servidor via JavaScript (por exemplo, imagens, texto, JSON e até trechos de código HTML), o que significa que podemos atualizar pequenas seções de conteúdo sem ter que recarregar toda página. Isso levou a páginas da Web mais responsivas e parece empolgante, mas está além do escopo deste artigo ensinar isso com muito mais detalhes³.

Para começar, vamos armazenar a URL do JSON que queremos recuperar em uma variável. Adicione o seguinte na parte inferior do seu código JavaScript:

```
1  let requestURL =  
    'https://raw.githubusercontent.com/luizpicolo/json-for-testing  
2  /main/dragonball.json';
```

Para criar uma solicitação, precisamos criar uma nova instância de objeto de solicitação a partir do construtor **XMLHttpRequest** usando a palavra-chave **new**. Adicione o seguinte abaixo sua última linha:

```
1  let request = new XMLHttpRequest();
```

Agora precisamos abrir uma nova solicitação usando o método **open()**. Adicione a seguinte linha:

```
1  request.open('GET', requestURL);
```

³ Para mais detalhes acesse: <<https://developer.mozilla.org/pt-BR/docs/Learn/JavaScript/Objects/JSON>>

Isso leva pelo menos dois parâmetros — existem outros parâmetros opcionais disponíveis. Nós só precisamos dos dois obrigatórios para este exemplo simples:

- O método HTTP a ser usado ao fazer a solicitação de rede. Neste caso, GET é bom, pois estamos apenas recuperando alguns dados simples.
- O URL para fazer a solicitação — esta é a URL do arquivo JSON que armazenamos anteriormente.

Em seguida, adicione as duas linhas a seguir — aqui estamos definindo o `responseType` como JSON, para que o XHR saiba que o servidor retornará o JSON e que isso deve ser convertido nos bastidores em um objeto JavaScript. Em seguida, enviamos a solicitação com o método `send()`:

```
1 request.responseType = 'json';
2 request.send();
```

A última parte desta seção envolve aguardar a resposta retornar do servidor e, em seguida, lidar com ela. Adicione o seguinte código abaixo do seu código anterior:

```
1 request.onload = function() {
2   let personagens = request.response;
3
4   const elemento = document.getElementById('list');
5   let p1 = personagens.characters[0].name;
6   elemento.insertAdjacentHTML('afterbegin', p1);
7 }
```

3.4 Exercícios de Fixação

- A partir do arquivo **noticias.json** criado no exercício anterior, e utilizando também as classes **Noticia** e **NoticiaDestaque** implementadas, leia e apresente no navegador as notícias contidas no arquivo **noticias.json**. Para tanto, você deve implementar a leitura do arquivo e, a partir dos dados, criar objetos e apresentar os mesmos no navegador.

4 Versionamento de Código

Um ladrão rouba um tesouro, mas não furta a inteligência.

Uma crise destrói um herança, mas não uma profissão.

*Não importa se você não tem dinheiro, você é uma pessoa rica,
pois possui o maior de todos os capitais: a sua inteligência.*

*Invista nela. **Estude!***

Augusto Cury

De forma simples, podemos dizer que o controle de versão é um sistema que registra as mudanças feitas em um arquivo ou um conjunto de arquivos ao longo do tempo de forma que você possa recuperar versões específicas para praticamente qualquer tipo de arquivo em um computador.

Assim, esse capítulo, tem como objetivo proporcionar a você leitor(a) os pontos os quais julgamos mais importantes para compreender e praticar o controle de versão. Esse tipo de técnica é amplamente utilizado em ambientes de trabalho, nos quais, várias pessoas tem que trabalhar em um mesmo projeto. Assim, caso um erro aconteça, ou características nova surjam, os membros da equipe podem corrigir o problema de forma simples e rápida.

4.1 Projeto de banco de dados

Referências

FARINELLI, F. Conceitos basicos de programação orientada a objetos. *Instituto Federal Sudeste de Minas Gerais*, 2007. Citado na página 2.

HAVERBEKE, M. *Eloquent JavaScript: a modern introduction to programming*. [S.l.]: No Starch Press, 2018. Citado na página 23.

PRESSMAN, R.; MAXIM, B. *Engenharia de Software-8ª Edição*. [S.l.]: McGraw Hill Brasil, 2016. Citado na página 4.

ROY, P. V. Programming paradigms for dummies: What every programmer should know. 04 2012. Citado na página 3.

RUIZ, E. E. S. *IBm1030 Programação Orientada a Objetos*. 2008. Disponível em: <http://dcm.ffclrp.usp.br/~evandro/ibm1030/constru/heranca.html>. Citado 2 vezes nas páginas 7 e 22.

SILVA, D. Lucas da et al. Ontologias e unified modeling language: uma abordagem para representação de domínios de conhecimento. v. 10, 10 2009. Citado na página 9.

SOMMERVILLE, I. *Engenharia de Software. 6ª*. [S.l.: s.n.], 2003. Citado 3 vezes nas páginas 2, 3 e 4.

TRASVIÑA, F. *Introdução ao JavaScript Orientado a Objeto*. 2021. Disponível em: <https://developer.mozilla.org/pt-BR/docs/conflicting/Learn/JavaScript/Objects>. Nenhuma citação no texto.