PROGRAMAÇÃO ORIENTADA A OBJETOS

(AULA 19)

CURSO BÁSICO DE PROGRAMAÇÃO COM JAVASCRIPT

MAYARA MARQUES

mmrosatab@gmail.com

JS

SUMÁRIO

- Recordando
- Abstração
- Encapsulamento
- Herança
- Polimorfismo
- Mão na massa

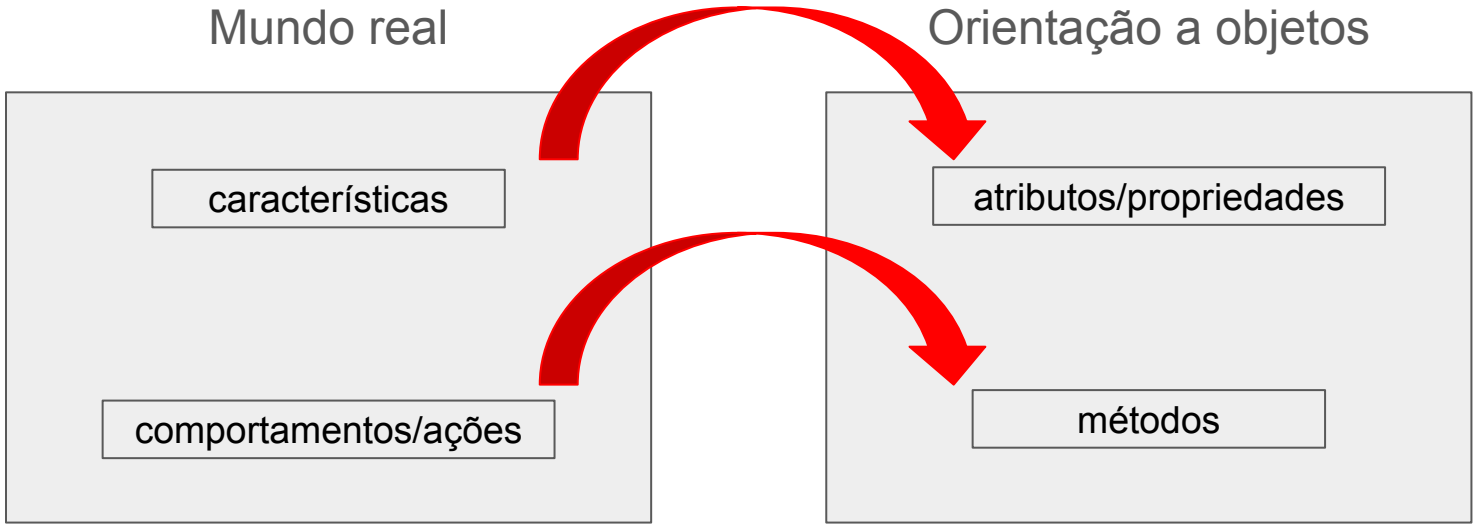
RECORDANDO...

Orientação a objetos é um ***paradigma de programação*** que se baseia no conceito de ***objetos*** para ***representar os seres e coisas do mundo real***.

Objetos são entidades que representam algo do mundo real, possuindo **características** e **comportamentos**.

RECORDANDO...

Dentro da orientação a objetos essas ***características*** e ***comportamentos*** são chamados respectivamente de ***atributos*** e ***métodos***.



PILARES DA ORIENTAÇÃO A OBJETOS

Os quatro pilares da programação orientada a objetos (POO) são: **Abstração**, **Encapsulamento**, **Herança** e **Polimorfismo**

Nesta aula, abordaremos sobre cada um deles.

ABSTRAÇÃO

Abstração é a ideia de representar um objeto focando só no que é importante para o problema, escondendo os detalhes internos que não precisam ser conhecidos por quem usa esse objeto.

ABSTRAÇÃO

JS

Minimundo

Imagine que você está criando um programa para uma locadora de carros. Cada carro precisa ter uma marca para ser identificado, como "Toyota" ou "Honda". Também precisamos saber a velocidade atual de cada carro, que começa em zero. E, claro, queremos poder acelerar o carro para aumentar sua velocidade.

ABSTRAÇÃO

Minimundo

Imagine que você está criando um programa para uma locadora de carros. Cada ***carro*** precisa ter uma ***marca*** para ser identificado, como "Toyota" ou "Honda". Também precisamos saber a ***velocidade*** atual de cada carro, que começa em zero. E, claro, queremos poder ***acelerar*** o carro para aumentar sua velocidade.

ABSTRAÇÃO

Através da análise do contexto apresentado e da abstração, podemos criar uma classe como a mostrada abaixo:

```
class Car {
  constructor(brand) {
    this.brand = brand      // marca do carro
    this.speed = 0          // velocidade inicial
  }

  accelerate() {
    this.speed = this.speed + 10
    console.log(`${this.brand} accelerated to ${this.speed} km/h`)
  }
}
```

ENCAPSULAMENTO

Encapsulamento é o conceito de esconder os detalhes de implementação de uma classe e expor apenas o que é necessário para o seu uso externo.

ENCAPSULAMENTO

O encapsulamento tem como um dos objetivos permitir **um maior controle sobre como os dados de objetos são expostos** e **como podem ser modificados**.

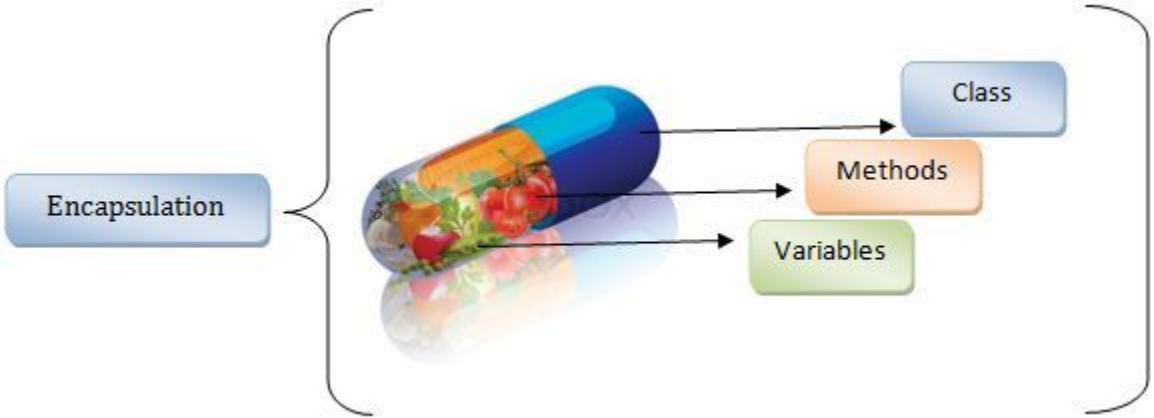


Imagem extraída de: https://logicmojo.com/encapsulation-in-oops#2

ENCAPSULAMENTO

O código abaixo especifica um modelo para um carro e cria uma instância de carro.

```
class Car {
  constructor(brand) {
    this.brand = brand
    this.speed = 0
  }
}

const car = new Car("Uno")
```

ENCAPSULAMENTO

Como podemos ver no código abaixo, não há nenhuma barreira ou proteção que impeça alterar o valor da velocidade do carro. Pode-se até colocar um valor negativo.

```
class Car {
  constructor(brand) {
    this.brand = brand      // marca do carro
    this.speed = 0          // velocidade inicial
  }
}

const car = new Car("Uno")
car.speed = -1000 // Isso não deveria ser permitido!
console.log(car.speed) // -1000
```

ENCAPSULAMENTO

Propriedades privadas

Podemos tornar uma propriedade **privada** em uma classe. Em outras palavras, podemos restringir o acesso direto a esse valor. Para isso, utiliza-se o símbolo **#** à frente da propriedade que se deseja restringir.

```
class Car {

  constructor(brand) {
    this.#brand = brand
    this.#speed = 0
  }
}
```

ENCAPSULAMENTO

O código abaixo resultará em erro, visto que o JavaScript protege campos declarados com #. Eles **não podem ser acessados nem modificados diretamente fora da classe**.

```
class Car {
  #speed
  #brand
  constructor(brand) {
    this.#brand = brand
    this.#speed = 0
  }
}

const car = new Car("Uno")
car.#speed = -1000
console.log(car.#speed)
```

```
ERROR!
/tmp/kFCNS6WMbK/main.js:9
car.#speed = -1000;
  ^

SyntaxError: Private field '#speed' must
be declared in an enclosing class
```

ENCAPSULAMENTO

Mas se restringirmos o acesso direto, como faremos para atualizar o valor de uma propriedade?

ENCAPSULAMENTO

Podemos utilizar **métodos específicos** tanto para fazer a atualização do valor de uma propriedade, quanto para obter o valor dessa propriedade.

Para isso, utilizamos métodos chamados **getters** e **setters**.

ENCAPSULAMENTO

Getter

Getter é uma função usada para **pegar (ler)** o valor de uma propriedade de forma controlada.

Setter

Setter é uma função usada para **definir (alterar)** o valor de uma propriedade de forma segura.

```
class Car {
        #brand
        #speed

        constructor(brand) {
                this.#brand = brand
                this.#speed = 0
        }

        // Getter para acessar a marca
        get brand() {
                return this.#brand
        }

        // Setter para alterar a marca (com validação opcional)
        set brand(value) {
                if (typeof value === 'string' && value.trim() !== '') {
                        this.#brand = value
                } else {
                        console.log("Invalid brand. Must be a non-empty string.")
                }
        }

        // Getter para acessar a velocidade
        get speed() {
                return this.#speed
        }

        // Setter para alterar a velocidade (com validação)
        set speed(value) {
                if (typeof value === 'number' && value >= 0) {
                        this.#speed = value
                } else {
                        console.log("Invalid speed. Must be a non-negative number.")
                }
        }

        accelerate() {
                this.#speed = this.#speed + 10
                console.log(`${this.#brand} accelerated to ${this.#speed} km/h`)
        }
}
```

Veja que, a partir de agora, nós conseguimos alterar o valor de speed via **set**, mas sob determinadas condições. O valor deverá ser um número e deve ser maior ou igual a zero.

ENCAPSULAMENTO

Apesar de termos os **métodos get** e **set** definidos da mesma maneira como definimos outros métodos. No momento de chamá-los, continuaremos **a usar a sintaxe como se fossem atributos**

```
// Exemplo de uso
const car = new Car("Uno")

console.log(car.speed) // 0

car.speed = 120
console.log(car.speed) // 120
```

Set

Get

ENCAPSULAMENTO

```
// Exemplo de uso
const car = new Car("Uno")

console.log(car.speed) // 0

car.speed = 120
console.log(car.speed) // 120

car.speed = -50        // Invalid speed. Must be a non-negative number.
console.log(car.speed)  // 120
```

```
class Car {
        #brand
        #speed

        constructor(brand) {
                this.#brand = brand
                this.#speed = 0
        }

        // Getter para acessar a marca
        get brand() {
                return this.#brand
        }

        // Setter para alterar a marca (com validação opcional)
        set brand(value) {
                if (typeof value === 'string' && value.trim() !== '') {
                        this.#brand = value
                } else {
                        console.log("Invalid brand. Must be a non-empty string.)"
                }
        }

        // Getter para acessar a velocidade
        get speed() {
                return this.#speed
        }

        // Setter para alterar a velocidade (com validação)
        set speed(value) {
                if (typeof value === 'number' && value >= 0) {
                        this.#speed = value
                } else {
                        console.log("Invalid speed. Must be a non-negative number.)"
                }
        }

        accelerate() {
                this.#speed = this.#speed + 10
                console.log(`${this.#brand} accelerated to ${this.#speed} km/h`)
        }
}
```

Normalmente, chamamos métodos assim. Contudo, essa maneira não é utilizada no JavaScript para os métodos get e set. Estes são chamados como se fossem acesso/atualização de uma propriedade.

```
const c1 = new Car("Uno")

c1.accelerate()              // Uno acelerou para 10 km/h

console.log(c1.brand)        // Get > Exibe: Uno
c1.brand = "Palio"
console.log(c1.brand)       // Get > Exibe: Palio
c1.brand = ""              // Set > Marca inválida. Deve ser uma string não vazia.
```

ENCAPSULAMENTO

Por que encapsular?

Proteção de dados: O encapsulamento impede que outras partes do código modifiquem ou acessem diretamente os atributos de um objeto, evitando erros e inconsistências.

Flexibilidade: Permite que a implementação interna de um objeto seja alterada sem afetar o código que o utiliza.

Modularidade: Ao ocultar detalhes internos, os objetos se tornam mais independentes e fáceis de reutilizar em diferentes partes do sistema.

Manutenção: O encapsulamento simplifica a manutenção do código, pois permite modificar a implementação interna sem ter que alterar todo o código que utiliza o objeto.

Segurança: Ajuda a proteger dados sensíveis ou críticos de um objeto, impedindo que **sejam acessados ou alterados indevidamente**.

HERANÇA

Herança é um princípio da programação orientada a objetos onde uma classe pode "**herdar**" **propriedades** e **métodos** de outra classe.

Isso permite **reutilizar código**, **evitar repetição** e **criar especializações**.

```
class Car {
  #brand
  #speed

  constructor(brand) {
    this.#brand = brand
    this.#speed = 0
  }

  get brand() {
    return this.#brand
  }

  set brand(value) {
    if (typeof value === 'string' && value.length > 0) {
      this.#brand = value
    } else {
      console.log("Invalid brand name")
    }
  }

  get speed() {
    return this.#speed
  }

  set speed(value) {
    if (value >= 0) {
      this.#speed = value
    } else {
      console.log(`Invalid speed for ${ this.#brand}`)
    }
  }

  accelerate() {
    this.#speed = this.#speed + 10
    console.log(`${this.#brand} accelerated to ${ this.#speed} km/h`)
  }
}
```

```
class ElectricCar extends Car {
  #batteryLevel

  constructor(brand, batteryLevel) {
    super(brand)
    this.#batteryLevel = batteryLevel
  }

  get battery() {
    return this.#batteryLevel
  }

  set battery(value) {
    if (value >= 0 && value <= 100) {
      this.#batteryLevel = value
    } else {
      console.log(`Invalid battery level for ${ this.brand}`)
    }
  }

  charge() {
    this.#battery = 100
    console.log(`${this.#brand} was charged to ${ this.#battery}%`)
  }
}
```

```
const tesla = new ElectricCar("Tesla", 80)

tesla.accelerate()       // método herdado que usa o speed encapsulado
tesla.charge()           // método da subclasse

console.log(tesla.brand)      // getter da classe Car
console.log(tesla.speed)      // getter da classe Car
console.log(tesla.battery)    // getter da classe ElectricCar

tesla.brand = ""              // tentativa inválida de alterar a marca → mensagem de erro
tesla.speed = -50             // tentativa inválida de alterar a velocidade → mensagem de erro
tesla.battery = 150           // tentativa inválida de alterar a bateria → mensagem de erro
```

HERANÇA

“Protótipos são o mecanismo pelo qual objetos JavaScript herdam recursos uns dos outros.”

Texto extraído de https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Extensions/Advanced_JavaScript_objects/Object_prototypes

PROTÓTIPOS

Para fornecer herança, o JavaScript permite que um objeto tenha um outro objeto dentro dele que não foi adicionado explicitamente no momento de sua construção.

Esse objeto contém métodos e atributos que são de uma classe pai. A este objeto se dá o nome de objeto protótipo.

PROTÓTIPOS

"O JavaScript é frequentemente descrito como uma linguagem baseada em protótipos — para fornecer herança, os objetos podem ter um objeto de protótipo, que atua como um objeto de modelo do qual herda métodos e propriedades. O objeto de protótipo de um objeto também pode ter um objeto de protótipo, do qual herda métodos e propriedades, e assim por diante. Isso geralmente é chamado de cadeia de protótipos e explica por que objetos diferentes têm propriedades e métodos definidos em outros objetos disponíveis para eles."

Texto extraído de https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Extensions/Advanced_JavaScript_objects/Object_prototypes

POLIMORFISMO

Polimorfismo significa "muitas formas".

Em orientação a objetos, ele permite que **métodos com o mesmo nome se comportem de formas diferentes**, dependendo da classe onde estão implementados.

POLIMORFISMO

```
class Car {
  #speed // campo privado

  constructor(brand) {
    this.brand = brand
    this.#speed = 0
  }

  // Getter para velocidade
  get speed() {
    return this.#speed
  }

  // Setter para velocidade
  set speed(value) {
    if (value >= 0) {
      this.#speed = value
    }
  }

  // Método de aceleração
  accelerate() {
    this.#speed += 10
    console.log(`${this.#brand} accelerated to ${ this.#speed} km/h`)
  }
}
```

POLIMORFISMO

```
class ElectricCar extends Car {
  #battery // campo privado

  constructor(brand, battery) {
    super(brand)
    this.#battery = battery
  }

  // Getter para bateria
  get battery() {
    return this.#battery
  }

  // Setter para bateria
  set battery(value) {
    if (value >= 0 && value <= 100) {
      this.#battery = value
    }
  }

  // Polimorfismo: redefine o método accelerate
  accelerate() {
    this.speed += 15
    this.battery -= 1
    console.log(`${this.brand} (electric) accelerated to ${ this.speed} km/h with ${ this.battery}%
battery`)
  }
}
```

POLIMORFISMO

```
// Instâncias de exemplo
const uno = new Car("Fiat")
const tesla = new ElectricCar("Tesla", 80)

uno.accelerate()      // Fiat accelerated to 10 km/h
tesla.accelerate()    // Tesla (electric) accelerated to 15 km/h with 79% battery
```

POLIMORFISMO

Polimorfismo dinâmico acontece quando um método é sobrescrito por subclasses e o comportamento é decidido em tempo de execução, dependendo do tipo real do objeto.

POLIMORFISMO

```
class Document {
  print() {
    console.log("Imprimindo documento genérico...")
  }
}

class Report extends Document {
  print() {
    console.log("Imprimindo relatório com gráficos e tabelas.")
  }
}

class Contract extends Document {
  print() {
    console.log("Imprimindo contrato com cláusulas e assinaturas.")
  }
}

class Letter extends Document {
  print() {
    console.log("Imprimindo carta com saudação e assinatura.")
  }
}
```

POLIMORFISMO

```
const docs = [
  new Report(),
  new Contract(),
  new Letter()
]

docs.forEach(doc => {
  doc.print()
})
```

```
Imprimindo relatório com gráficos e tabelas.
Imprimindo contrato com cláusulas e assinaturas.
Imprimindo carta com saudação e assinatura.
```

MÃO NA MASSA

MÃO NA MASSA

1. Crie um modelo de pessoa que possua um nome e uma idade.

Depois, implemente um método que mostre a seguinte frase no console:

"Hi, my name is <name> and I’m <age> years old."

MÃO NA MASSA

2. Melhore a classe da pessoa que você criou.

Agora, os dados devem ficar inacessíveis diretamente de fora da classe.

Implemente uma forma segura de acessar e alterar essas informações.

Garanta que a idade nunca seja negativa.

MÃO NA MASSA

3. Crie uma conta bancária que começa com saldo zero.

Implemente funções para:

Depositar um valor

Sacar um valor (apenas se houver saldo suficiente)

Consultar o saldo atual

Simule as ações e veja os resultados no console.

MÃO NA MASSA

4. Crie um tipo genérico de animal, com um método que mostra um som.

Depois, crie dois tipos específicos de animal e altere esse comportamento.

Por fim, teste os dois animais e compare o resultado

MÃO NA MASSA

5. Você deve criar um tipo geral de veículo, com marca e um método para ligar.

Depois, crie dois tipos diferentes de veículos e altere a forma como eles ligam.

Por fim, escreva uma função que aceite qualquer um desses veículos e o ligue, mostrando o comportamento específico de cada um.

MÃO NA MASSA

6. Crie um carro que tenha velocidade e bateria, ambas com valores protegidos.

Garanta que a bateria nunca ultrapasse 100% nem fique negativa.

Adicione uma função para carregar a bateria.

Simule acelerar o carro e carregar a bateria.

MÃO NA MASSA

7. Crie uma estrutura de mensagem com remetente, destinatário e conteúdo.

Garanta que nenhuma mensagem seja enviada com conteúdo vazio.

Mostre a mensagem formatada no console ao enviá-la.

MÃO NA MASSA

8. Crie um produto com nome e preço.

Adicione uma forma de aplicar descontos.

Agora, crie uma versão digital do produto que sempre tenha um desconto fixo.

Simule a compra de produtos diferentes e exiba o preço final com desconto.

REFERÊNCIAS

- http://developer.mozilla.org/pt-BR/docs/Learn_web_development/Extensions/Advanced_JavaScript_objects
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set
- https://developer.mozilla.org/pt-BR/docs/Learn_web_development/Extensions/Advanced_JavaScript_objects/Classes_in_JavaScript
- https://www.geeksforgeeks.org/java/difference-between-method-overloading-and-method-overriding-in-java/