1. Classe e Objeto

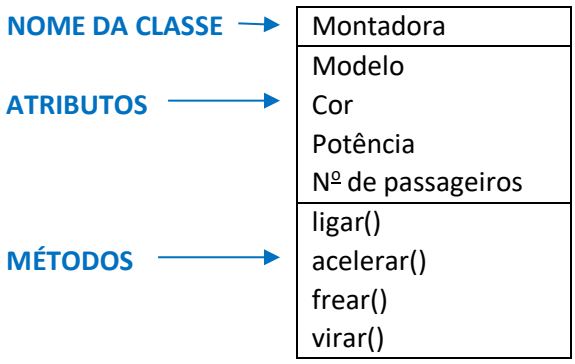
Neste capítulo vamos abordar a programação orientada a objeto também conhecida pela sigla POO. Tradicionalmente a programação orientada a objeto inicia-se com a elaboração de uma classe, que nada mais é que um projeto de um determinado objeto. Para elaborar essa classe precisamos conhecer os atributos ou variáveis de instância desse objeto e quais são as suas ações ou métodos. Essas informações são a essência do objeto, pois serão utilizadas para a criação dele.

Para entender melhor os conceitos de POO vamos abstrair e pensar em uma montadora de carros. A fábrica representa a Classe, pois é ela quem produz os carros que, neste caso, serão os nossos objetos. Normalmente uma fábrica produz vários carros, ou seja, quando temos uma classe ela deve ser capaz de produzir vários objetos.

Quando falamos do carro uma ideia já se forma na sua cabeça: tem quatro rodas, motor, vidros, bancos, direção, espelho, faróis e etc. Essas características são comuns a todos os carros e definem o objeto carro, mas existem características tais com: modelo, cor, potência do motor, número de passageiros, acessórios e etc. Neste caso estamos falando de características especificas de cada um dos carros fabricados, ou seja, são elas que diferenciam um carro do outro, isto é o que chamamos de atributos do nosso objeto.

Existem várias ações ligadas ao objeto carro, tais como: ligar, acelerar, frear, andar para frente, andar para traz, virar à esquerda, virar à direita e etc. As ações ligadas ao objeto são os métodos da nossa classe. Dentro de uma classe os métodos são definidos por funções.

Resumidamente podemos definir a classe como a estrutura básica de uma coleção de objetos. Os objetos são os elementos criados com base nessa estrutura, os atributos são as características específicas de cada um dos objetos e os métodos são as ações dos objetos. Abaixo podemos ver um exemplo de classe associada a uma fábrica de carros:



1.1 Criando objetos

Para o Java Script o objeto é um conjunto de pares chave e valor. A maneira mais simples de criar um objeto é definir uma variável e acrescentar o conjunto de atributos do objeto.

```
const carro = {marca : "Fiat", cor: "verde", modelo: "Palio", ano: 2018}
```

Podemos acessar cada um dos atributos a partir do operador ponto.

```
1   const carro = {marca : "Fiat", cor: "verde", modelo: "Palio", ano: 2018}
2
3   console.log(carro.marca)
4   console.log(carro.cor)
5   console.log(carro.modelo)
6   console.log(carro.ano)
```

```
[Running] node "c:\Users\
Fiat
verde
Palio
2018
```

Da mesma forma que exibimos o valor de um atributo também podemos alterar seu valor.

```
1   const carro = {marca : "Fiat", cor: "verde", modelo: "Palio", ano: 2018}
2
3   console.log(carro.cor)
4   carro.cor = "azul"
5   console.log(carro.cor)
```

```
[Running] node "c:\Users'
verde
azul
```

Os atributos não precisam ser atribuídos todos na declaração do objeto, eles podem ser adicionados dinamicamente.

```
1   const carro = {marca : "Fiat", cor: "verde", modelo: "Palio", ano: 2018}
2   carro.valor = "R$ 30000,00"
3   console.log(carro)
```

```
[Running] node "c:\Users\Fer
{
  marca: 'Fiat',
  cor: 'verde',
  modelo: 'Palio',
  ano: 2018,
  valor: 'R$ 30000,00'
}
```

Os atributos também podem ser excluídos de um objeto, para tanto utilizamos o comando **delete**.

```
1   const carro = {marca : "Fiat", cor: "verde", modelo: "Palio", ano: 2018}
2   console.log(carro)
3   delete carro.modelo
4   console.log(carro)
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\React Native'
{ marca: 'Fiat', cor: 'verde', modelo: 'Palio', ano: 2018 }
{ marca: 'Fiat', cor: 'verde', ano: 2018 }
```

O atributo de um objeto pode ser um outro objeto.

```
1   const carro = {marca : "Fiat", modelo: "Palio",
2         especif: {cor: "verde", ano: 2018, valor: "R$3000"}}
3   console.log(carro)
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\React Na
{
  marca: 'Fiat',
  modelo: 'Palio',
  especif: { cor: 'verde', ano: 2018, valor: 'R$3000' }
}
```

Para acessar os atributos de **especif** primeiro deve ser referenciado o nome do objeto principal, depois o nome do objeto secundário e finalmente o nome do atributo.

```
1   const carro = {marca : "Fiat", modelo: "Palio",
2         especif: {cor: "verde", ano: 2018, valor: "R$3000"}}
3   console.log(carro.especif.cor)
4   console.log(carro.especif.ano)
5   console.log(carro.especif.valor)
```

```
[Running] no
verde
2018
R$3000
```

Uma forma diferente que o Java Script utiliza para acessar um atributo é através de colchetes. Neste caso não utilizamos o ponto e utilizamos uma string com o nome do atributo dentro de colchetes.

```
1   const carro = {marca : "Fiat", modelo: "Palio",
2         especif: {cor: "verde", ano: 2018, valor: "R$3000"}}
3   console.log(carro['marca'])
4   console.log(carro['modelo'])
5   console.log(carro['especif']['cor'])
6   console.log(carro['especif']['ano'])
7   console.log(carro['especif']['valor'])
```

```
[Running] node "c:\User
Fiat
Palio
verde
2018
R$3000
```

Para criarmos um objeto vazio podemos simplesmente definir o nome seguido das chaves de abertura e fechamento.

```
1   const carro = {}
2
3   console.log(typeof(carro))
```

```
[Running] node "c:\
object
```

Lógico que neste caso os atributos devem ser atribuídos a posterior.

```
const carro = {}
carro.marca = "Fiat"
carro.modelo = "Palio"
carro.especif = {}
carro.especif.cor = "verde"
carro.especif.ano = 2018
carro.especif.valor = "R$3000"

console.log(carro)
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\React Nat
{
  marca: 'Fiat',
  modelo: 'Palio',
  especif: { cor: 'verde', ano: 2018, valor: 'R$3000' }
}
```

Observe que embora o objeto tenha sido definido com **const** nós conseguimos alterar e incluir atributos, isso acontece porque estamos acessando elementos do objeto. Caso tente fazer uma atribuição de valores para o objeto inteiro será exibido uma mensagem de erro.

```
const carro = {}
carro = {marca : "Fiat", modelo: "Palio",
        especif: {cor: "verde", ano: 2018, valor: "R$3000"}}

console.log(carro)
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\React Nat
c:\Users\Ferrara\OneDrive\Docs\React Native\teste.js:2
carro = {marca : "Fiat", modelo: "Palio",
      ^

TypeError: Assignment to constant variable.
```

Este erro pode ser evitado se declararmos o objeto com **let** ou **var** .

```
let carro = {}
carro = {marca : "Fiat", modelo: "Palio",
        especif: {cor: "verde", ano: 2018, valor: "R$3000"}}

console.log(carro)
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\React Nat
{
  marca: 'Fiat',
  modelo: 'Palio',
  especif: { cor: 'verde', ano: 2018, valor: 'R$3000' }
}
```

Outra forma de definir um objeto vazio é através do construtor **new Object**.

```
let carro = new Object

console.log(typeof(carro))
```

```
[Running] node "c:\L
object
```

O Java Script permite a utilização de arrays como conteúdo de atributos do objeto.

```
let churrasco = {
    carnes: ["maminha", "alcatra","picanha","costela"],
    acompanhamentos: ["arroz", "farofa", "vinagrete"],
    bebidas: ["cerveja", "refrigerante", "batida"]
}

console.log(churrasco)
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\React Nativ
{
  carnes: [ 'maminha', 'alcatra', 'picanha', 'costela' ],
  acompanhamentos: [ 'arroz', 'farofa', 'vinagrete' ],
  bebidas: [ 'cerveja', 'refrigerante', 'batida' ]
}
```

Você pode acessar o array inteiro através do atributo ou um elemento específico utilizando seu índice.

```
let churrasco = {
    carnes: ["maminha", "alcatra","picanha","costela"],
    acompanhamentos: ["arroz", "farofa", "vinagrete"],
    bebidas: ["cerveja", "refrigerante", "batida"]
}

console.log(churrasco.carnes)
console.log(churrasco.bebidas[1])
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\Re
[ 'maminha', 'alcatra', 'picanha', 'costela' ]
refrigerante
```

1.2 Criando métodos para o objeto

Os métodos são ações vinculadas ao nosso objeto e normalmente são definidos através de funções que pode ou não receber valores como parâmetro.

Inicialmente vamos criar um objeto que representa o histórico de um aluno em determinada matéria. Definimos como atributos o ra, o nome, as notas das provas P1 e P2, a média, as faltas e o estado na matéria.

```
let aluno ={
    ra: "1234",
    nome: "João",
    p1: 6.0,
    p2: 5.5,
    media: null,
    faltas: 0,
    estado: "NC",
}
```

Agora vamos implementar um método para calcular a média do aluno.

```
let aluno ={
    ra: "1234",
    nome: "João",
    p1: 6.0,
    p2: 5.5,
    media: null,
    faltas: 0,
    estado: "NC",
    calcMedia(){
        this.media = (this.p1+2*this.p2)/3
        this.media = Number(this.media.toFixed(2))
    }
}

console.log(aluno.media)
aluno.calcMedia()
console.log(aluno.media)
```

```
[Running] node "c:\Use
null
5.67
```

Observe que exibimos o valor da média antes e depois de invocar o método, inicialmente seu valor era nulo e depois passou para 5,67.

Agora vamos fazer um método que sempre que for chamado acrescentara uma unidade ao número de faltas.

```
    calcMedia(){
        this.media = (this.p1+2*this.p2)/3
        this.media = Number(this.media.toFixed(2))
    },
    addfaltas() { this.faltas++ }
}

console.log(aluno.faltas)
aluno.addfaltas()
console.log(aluno.faltas)
aluno.addfaltas()
console.log(aluno.faltas)
```

```
[Running] node "c:\U
0
1
2
```

Para finalizar vamos implementar um método que atualiza o estado do aluno. Para seu estado mudar de “NC” obrigatoriamente a média não pode ser nula. Caso tenha uma média maior ou igual a 5,0 e possua no máximo 4 faltas ele estará “Aprovado” e caso contrário “Reprovado”

```
13      addfaltas() { this.faltas++ },
14      attEstado() {
15          if (this.media != null){
16              if ( this.media >=5 && this.faltas < 5){
17                  this.estado = "Aprovado"
18              }
19              else {this.estado = "Reprovado"}
20          }
21      }
22  }
23
24  console.log(aluno.estado)
25  aluno.calcMedia()
26  aluno.attEstado()
27  console.log(aluno.estado)
```

```
[Running] node
NC
Aprovado
```

1.3 Congelando Um Objeto

Nós já vimos que para criar um objeto podemos utilizar as palavras reservadas **let**, **var** ou **const**, seguidas do nome escolhido e de chaves. Quando utilizamos **const** o endereço de memória atribuído ao objeto não pode ser mudado, mas seus atributos internos sim.

```
1   const pessoa = { nome: "João", idade: 32}
2   console.log(pessoa)
3   pessoa.nome = "Maria"
4   console.log(pessoa)
```

```
[Running] node "c:\Users\Ferra
{ nome: 'João', idade: 32 }
{ nome: 'Maria', idade: 32 }
```

Neste caso se tentarmos atribuir um novo objeto a **pessoa** será gerado um erro.

```
1   const pessoa = { nome: "João", idade: 32}
2   console.log(pessoa)
3   pessoa = {nome = "Maria", idade: 25}
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\React
c:\Users\Ferrara\OneDrive\Docs\React Native\Fundamer
pessoa = {nome = "Maria", idade: 25}
         ^^^^^^^^^^^^^

SyntaxError: Invalid shorthand property initializer
```

Para impedir a mudança ou inclusão de atributos do objeto é necessário congelar o mesmo, para isso utilizamos o método **Object.freeze()**.

```
1  const pessoa = { nome: "João", idade: 32}
2  console.log(pessoa)
3  pessoa.nome = "Maria"
4  pessoa.idade = 25
5  console.log(pessoa)
6  Object.freeze(pessoa)
7  pessoa.nome = "José"
8  pessoa.idade = 18
9  console.log(pessoa)
```

```
[Running] node "c:\Users\Ferrara\(
{ nome: 'João', idade: 32 }
{ nome: 'Maria', idade: 25 }
{ nome: 'Maria', idade: 25 }
```

Como podemos ver, antes do objeto ser congelado conseguimos alterar seus atributos, após o congelamento isso não foi mais possível. Observe que não é gerada nenhuma mensagem de erro.

Para bloquear atributos individualmente os mesmos devem ser criados através do método **Object.defineProperty** , a utilização deste método e realizada depois que o objeto foi criado. Ele recebe como parâmetro o objeto, o nome do atributo a ser criado e conjunto de propriedades associadas ao atributo.

```
1  const pessoa = {nome: "João", idade:"32"}
2  Object.defineProperty(pessoa, 'cpf',{
3      writable: false,
4      enumerable: true,
5      value: "065157859-58"
6  })
7  console.log(pessoa)
8  pessoa.cpf = "123456789-36"
9  console.log(pessoa)
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\Reac
{ nome: 'João', idade: '32', cpf: '065157859-58' }
{ nome: 'João', idade: '32', cpf: '065157859-58' }
```

Como a opção **writable** foi definida como **false** o atributo não pode ser modificado e o valor atribuído através da propriedade **value** torna-se imutável.

A propriedade **enumerable** define se o atributo será visível ou não externamente, se for definida como **false** esta propriedade não será listada.

```
1  const pessoa = {nome: "João", idade:"32"}
2  Object.defineProperty(pessoa, 'cpf',{
3      writable: false,
4      enumerable: false,
5      value: "065157859-58"
6  })
7  console.log(pessoa)
```

```
[Running] node "c:\Users\Ferra
{ nome: 'João', idade: '32' }
```

Para acessar esta propriedade ela precisa ser requisitada textualmente.

```
const pessoa = {nome: "João", idade:"32"}
Object.defineProperty(pessoa, 'cpf',{
    writable: false,
    enumerable: false,
    value: "065157859-58"
})
console.log(pessoa)
console.log(pessoa.cpf)
```

```
[Running] node "c:\Users\Ferra
{ nome: 'João', idade: '32' }
065157859-58
```

1.4 Listando as chaves de um objeto

Para listar as chaves de um objeto devemos utilizar o método **Object.keys** , neste caso será retornado um array com todas as chaves do objeto.

```
let pessoa = {
    nome: 'João',
    idade: 32,
    sexo: "m",
    nacionalidade: "Brasileira"
}

console.log(Object.keys(pessoa))
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Doc
[ 'nome', 'idade', 'sexo', 'nacionalidade' ]
```

Este método retorna as chaves principais do objeto indicado.

```
let pessoa = {
    nome: 'João',
    idade: 32,
    sexo: "m",
    nacionalidade: "Brasileira",
    endereco:{
        rua: "Oswaldo Cruz",
        numero: 277,
        bairro: "Boqueirão",
        ciadede: "Santos"
    }
}

console.log(Object.keys(pessoa))
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\React Nati
[ 'nome', 'idade', 'sexo', 'nacionalidade', 'endereco' ]
```

Para listarmos as chaves do atributo **endereco** o mesmo de ser nominado textualmente.

```
        ciadede: "Santos"
    }
}

console.log(Object.keys(pessoa))
console.log(Object.keys(pessoa.endereco))
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\React Nativ
[ 'nome', 'idade', 'sexo', 'nacionalidade', 'endereco' ]
[ 'rua', 'numero', 'bairro', 'ciadede' ]
```

1.5 Listando os valores de um objeto

Para listar os valores de um objeto devemos utilizar o método **Object.values** , neste caso será retornado um array com todos os valores dos atributos do objeto.

```
let pessoa = {
    nome: 'João',
    idade: 32,
    sexo: "m",
    nacionalidade: "Brasileira",
}

console.log(Object.values(pessoa))
```

```
[Running] node "c:\Users\Ferrara\On
[ 'João', 32, 'm', 'Brasileira' ]
```

Se um dos atributos tiver como valor um objeto, este será retornado completamente.

```
let pessoa = {
    nome: 'João',
    idade: 32,
    sexo: "m",
    nacionalidade: "Brasileira",
    endereco:{
        rua: "Oswaldo Cruz",
        numero: 277,
        bairro: "Boqueirão",
        ciadede: "Santos"
    }
}
```

```
[Running] node "c:\Users\Ferrara\
[
  'João',
  32,
  'm',
  'Brasileira',
  {
    rua: 'Oswaldo Cruz',
    numero: 277,
    bairro: 'Boqueirão',
    ciadede: 'Santos'
  }
]
```

Para acessarmos os valores do objeto interno o mesmo deve ser referenciado textualmente.

```
        bairro: "Boqueirão",
        ciadede: "Santos"
    }
}

console.log(Object.values(pessoa.endereco))
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\
[ 'Oswaldo Cruz', 277, 'Boqueirão', 'Santos' ]
```

1.6 Listando as chaves e os valores de um objeto

Para listar os pares de chave/valore de um objeto devemos utilizar o método **Object.entries** , neste caso será retornada uma matriz com arrays contendo as chaves e os valores de cada atributo do objeto.

```
let pessoa = {
    nome: 'João',
    idade: 32,
    sexo: "m",
    nacionalidade: "Brasileira",
}

console.log(Object.entries(pessoa))
```

```
[Running] node "c:\Users\Ferrara\OneDriv
[
  [ 'nome', 'João' ],
  [ 'idade', 32 ],
  [ 'sexo', 'm' ],
  [ 'nacionalidade', 'Brasileira' ]
]
```

Se um dos atributos tiver como valor um objeto, será retornado no seu array a chave e o objeto.

```
let pessoa = {
    nome: 'João',
    idade: 32,
    sexo: "m",
    nacionalidade: "Brasileira",
    endereco:{
        rua: "Oswaldo Cruz",
        numero: 277,
        bairro: "Boqueirão",
        ciadede: "Santos"
    }
}

console.log(Object.entries(pessoa))
```

```
[Running] node "c:\Users\Ferrara\OneDr
[
  [ 'nome', 'João' ],
  [ 'idade', 32 ],
  [ 'sexo', 'm' ],
  [ 'nacionalidade', 'Brasileira' ],
  [
    'endereco',
    {
      rua: 'Oswaldo Cruz',
      numero: 277,
      bairro: 'Boqueirão',
      ciadede: 'Santos'
    }
  ]
]
```

Para acessarmos os valores/chaves do objeto interno o mesmo deve ser referenciado textualmente.

```
9           bairro: "Boqueirão",
10          ciadede: "Santos"
11      }
12  }
13
14  console.log(Object.entries(pessoa.endereco))
```

```
[Running] node "c:\Users\Ferrara\
[
  [ 'rua', 'Oswaldo Cruz' ],
  [ 'numero', 277 ],
  [ 'bairro', 'Boqueirão' ],
  [ 'ciadede', 'Santos' ]
]
```

1.7 Concatenado diversos objetos

Utilize o método **Object.assign** quando você tem vários objetos e deseja juntar todos os atributos em um único objeto.

```
1  let objeto1 = { nome: "João", idade: 32}
2  let objeto2 = { sexo: "M", nacionalidade: "Brasileira"}
3  let objeto3 = {endereco:{rua: "Oswaldo Cruz", numero: 277}}
4  let pessoa = Object.assign(objeto1, objeto2, objeto3)
5  console.log(pessoa)
```

```
[Running] node "c:\Users\Ferrara\OneDrive\Docs\Reac
{
  nome: 'João',
  idade: 32,
  sexo: 'M',
  nacionalidade: 'Brasileira',
  endereco: { rua: 'Oswaldo Cruz', numero: 277 }
}
```

Se tivermos atributos iguais em mais de um objeto, os valores serão sobrescritos prevalecendo o último valor adicionado.

1.8 Função construtora

Quando vamos utilizar vários objetos do mesmo tipo é recomendável recorrer a uma função construtora para este tipo de tarefa. No exemplo a seguir vamos declarar uma função construtora e depois instanciar três objetos.

```
function Retangulo(base, altura){
    this.base = base
    this.altura = altura
}

ret1 = new Retangulo(10,5)
ret2 = new Retangulo(7,8)
ret3 = new Retangulo(15,10)

console.log(ret1)
console.log(ret2)
console.log(ret3)
```

```
[Running] node "c:\Users\Ferrara\One
Retangulo { base: 10, altura: 5 }
Retangulo { base: 7, altura: 8 }
Retangulo { base: 15, altura: 10 }
```

Observe que quando exibimos os objetos, primeiro é mostrada a classe dele (Retangulo) e depois o seu conteúdo.

Nas funções construtoras também podemos definir métodos que estarão disponíveis para todos os objetos instanciados.

```
function Retangulo(base, altura){
    this.base = base
    this.altura = altura
    this.calcarea = function(){
        this.area = this.altura*this.base
    }
}

ret1 = new Retangulo(10,5)
ret1.calcarea()
console.log("A área é de: " + ret1.area)
```

```
[Running] node "c:\Users\Fer
A área é de: 50
```

Nestes exemplos todos os atributos dos objetos são públicos, isso ocorre quando referenciamos eles através do **this**, mas é possível termos atributos visíveis apenas dentro do objeto. Observe o exemplo a seguir:

```
function Pagamento(nome, salario, faltas){
    this.nome = nome
    this.faltas = faltas
}

func1 = new Pagamento("João",3000,2)
console.log(func1.nome)
console.log(func1.faltas)
console.log(func1.salario)
```

```
[Running] node "c:\Users
João
2
undefined
```

Como podemos ver, embora **salario** tenha sido passado como parâmetro quando tentamos exibir seu valor, externamente ao objeto é considerado como indefinido. Internamente podemos utilizar normalmente o valor recebido. Vamos acrescentar um método que retorna o valor salário a ser pago.

```
1   function Pagamento(nome, salario, faltas){
2       this.nome = nome
3       this.faltas = faltas
4       this.calcPagamneto = () => {
5           let pagamento = (salario/30)*(30-faltas)
6           return pagamento
7       }
8   }
9
10  func1 = new Pagamento("João",3000,2)
11  console.log("O salário a ser pago é de : R$ " + func1.calcPagamneto().toFixed(2))
```

```
[Running] node "c:\Users\Ferrara\OneDrive\
O salário a ser pago é de : R$ 2800.00
```

1.9 Definindo Classes

As versões mais recentes do Java Script permitem a definição da classe de uma forma mais convencional, tornando esta tarefa semelhante ao encontrado em outras linguagens de programação. Para esta tarefa utilizamos a palavra reservada **class** e o método **constructor**.

```
1   class Retangulo{
2       constructor(altura, base){
3           this.altura = altura
4           this.base = base
5       }
6       calcArea() {
7           this.area = this.altura*this.base
8       }
9
10  }
11
12  ret1 = new Retangulo(10,20)
13  console.log("altura = " + ret1.altura)
14  console.log("base = " + ret1.base)
15  ret1.calcArea()
16  console.log("área = " + ret1.area)
```

```
[Running] node "c:\Use
altura = 10
base = 20
área = 200
```

1.10 Utilizando Getters

Vamos supor que você tem um atributo que está vinculado ao cálculo de outros atributos, por exemplo podemos mencionar a área do retângulo que depende da sua base e altura. Podemos definir este atributo no próprio método construtor vinculando a estes dois parâmetros, mas nada impede que ele seja alterado posteriormente com um valor incompatível.

```
class Retangulo{
    constructor(altura, base){
        this.altura = altura
        this.base = base
        this.area = this.altura*this.base
    }
}

ret1 = new Retangulo(10,20)
console.log("altura = " + ret1.altura)
console.log("base = " + ret1.base)
console.log("área = " + ret1.area)
ret1.area = 50
console.log("\naltura = " + ret1.altura)
console.log("base = " + ret1.base)
console.log("área = " + ret1.area)
```

```
[Running] node "c:\Us
altura = 10
base = 20
área = 200

altura = 10
base = 20
área = 50
```

Como podemos ver a área ficou incompatível com os valores da base e altura. Outro problema que poderíamos ter é que a área só é calculada quando instanciamos o objeto, desta forma se depois alterarmos qualquer um dos atributos envolvidos seu valor vai se tornar incompatível pois, não é atualizado automaticamente.

```
class Retangulo{
    constructor(altura, base){
        this.altura = altura
        this.base = base
        this.area = this.altura*this.base
    }
}

ret1 = new Retangulo(10,20)
console.log("altura = " + ret1.altura)
console.log("base = " + ret1.base)
console.log("área = " + ret1.area)
ret1.base = 50
console.log("\naltura = " + ret1.altura)
console.log("base = " + ret1.base)
console.log("área = " + ret1.area)
```

```
[Running] node "c:\User
altura = 10
base = 20
área = 200

altura = 10
base = 50
área = 200
```

Uma forma de evitar este tipo de situação é utilizando um **Getter**, com a sua utilização sempre que o atributo for requisitado ele é acessado através do retorno de um método, o qual pode ser vinculado ao cálculo desejado. No Java Script para criar um **Getter** utilizamos a palavra reservada **get**.

```
1   class Retangulo{
2       constructor(altura, base){
3           this.altura = altura
4           this.base = base
5       }
6       get area(){
7           return this.altura*this.base
8       }
9   }
10
11  ret1 = new Retangulo(10,20)
12  console.log("altura = " + ret1.altura)
13  console.log("base = " + ret1.base)
14  console.log("área = " + ret1.area)
15  ret1.base = 50
16  console.log("\naltura = " + ret1.altura)
17  console.log("base = " + ret1.base)
18  console.log("área = " + ret1.area)
```

```
[Running] node "c:\U
altura = 10
base = 20
área = 200

altura = 10
base = 50
área = 500
```

Observe que a forma de acessar o atributo externamente não foi alterada **ret1.area** , mas internamente ele é acessado através do método **get**. Desta forma o valor da área será sempre atualizado conforme os valores de base e altura.

1.11 Utilizando Setters

O método **Setter** quando definido é chamado toda vez que tentamos atribuir um valor ao atributo. No Java Script utilizamos a palavra reservada **set** para definir este método e normalmente é utilizado em conjunto com um método **Getter**, inclusive esta é a única situação em que o Java Script permite a sobreposição de métodos. Sua principal utilização é na consistência de valores fornecidos a um atributo.

Vamos supor que você vai criar um objeto para armazenar dia e o mês de uma data, para garantir que não sejam atribuídos valores inconsistentes devemos recorrer a métodos do tipo **Setter**.

```
class Data{
    constructor(){
        this._dia = 1
        this._mes = 1
    }
    get dia(){return this._dia}
    get mes(){return this._mes}
    set dia(d){
        if (d > 0 && d < 32){
            this._dia = d
        }
    }
    set mes(m){
        if (m > 0 && m < 13){
            this._mes = m
        }
    }
}

const data1 = new Data(40,7)
data1.dia = 13
data1.mes = 10
console.log(data1.dia + "/" + data1.mes)
data1.dia = 20
data1.mes = 14
console.log(data1.dia + "/" + data1.mes)
data1.dia = 45
data1.mes = 11
console.log(data1.dia + "/" + data1.mes)
```

```
[Running] node "c:
13/10
20/10
20/11
```

Como podemos observar toda vez em que se tentou atribuir um valor fora do intervalo permitido ele não foi registrado. Outro detalhe importante é que foram criados os atributos auxiliares **_dia** e **_mes** para construção dos métodos envolvidos.

1.12 Herança em Java Script

A herança em Java Script é um pouco diferente das linguagens convencionais, ela trabalha com o conceito de protótipo. Toda função criada tem um protótipo responsável por sua criação, como arrays, objetos e strings para o Java Script são funções eles têm protótipos associados à sua criação. Esta característica é o que permite termos métodos nativos associados a eles.

Quando vinculamos o protótipo de um objeto ao de outro ele herda suas características, isto pode ser realizado acessando a propriedade **__proto__** . Inicialmente vamos mostrar como isto acontece com objetos literais, observe o exemplo a seguir:

```
1   const avo =  { nome: "João", sobrenome: "Silva", idade: 60}
2   const pai = { __proto__: avo, nome: "José", idade: 35, profissao: "Médico"}
3   const filho = {__proto__: pai, nome: "Pedro", idade: 12, sexo:"M"}
4
5   console.log("Nome do avo: " + avo.nome)
6   console.log("Nome do pai: " + pai.nome)
7   console.log("Nome do filho: " + filho.nome)
8
9   console.log("Sobrenome do avo: " + avo.sobrenome)
10  console.log("Sobrenome do pai: " + pai.sobrenome)
11  console.log("Sobrenome do filho: " + filho.sobrenome)
```

```
[Running] node "c:\Users\Fer
Nome do avo: João
Nome do pai: José
Nome do filho: Pedro
Sobrenome do avo: Silva
Sobrenome do pai: Silva
Sobrenome do filho: Silva
```

Como podemos observar como nome é um atributo que todos os objetos possuem, foi exibido os valores individuais correspondentes. No caso do sobrenome foi exibido “Silva” para todos, como o objeto **pai** não possuía este atributo foi procurado no objeto ao qual seu protótipo está vinculado, no caso o **avo**, como lá existia foi considerado este valor. Para o objeto **filho** após não ser encontrado em seus atributos, buscou-se no objeto **pai** com quem mantinha ligação, como novamente não foi encontrado buscou no objeto avo com quem o **pai** tinha ligação.

O Java Script trabalha a herança sempre desta forma, primeiro procura na instância local e caso não encontre vai procurando na cadeia de vínculos na qual o objeto está inserido. O mesmo procedimento pode ser aplicado para os métodos.

```
1   const avo =  { nome: "João", sobrenome: "Silva", idade: 60,
2       fazAniverario() { this.idade++}}
3   const pai = { __proto__: avo, nome: "José", idade: 35, profissao: "Médico"}
4   const filho = {__proto__: pai, nome: "Pedro", idade: 12, sexo:"M"}
5
6
7   console.log("idade do filho: " + filho.idade)
8   console.log("idade do pai: " + pai.idade)
9   filho.fazAniverario()
10  pai.fazAniverario()
11  console.log("idade do filho: " + filho.idade)
12  console.log("idade do pai: " + pai.idade)
```

```
[Running] node "c:\Users\Fer
idade do filho: 12
idade do pai: 35
idade do filho: 13
idade do pai: 36
```

Quando trabalhamos com classe ao invés do objeto literal a herança é um pouco diferente, neste caso na definição da classe indicamos de quem ela vai herdar atributos e métodos através da palavra reservada **extends**.

```
class Avo {
    constructor(nome, sobrenome, idade){
        this.nome = nome
        this.sobrenome = sobrenome
        this.idade = idade

    }
    fazAniverario() { this.idade++}
}

class Pai extends Avo {
    constructor(nome, sobrenome, idade, profissao){
        super(nome, sobrenome, idade)
        this.profissao = profissao
    }
}

pessoa = new Pai("João","Silva",40,"professor")
console.log(pessoa)
```

```
[Running] node "c:\Users\Ferr
Pai {
  nome: 'João',
  sobrenome: 'Silva',
  idade: 40,
  profissao: 'professor'
}
```

Observe que para os atributos herdados acessamos o método construtor da classe **Avo** através do comando **super()**. Os métodos da classe também são herdados como vemos a seguir:

```
        this.profissao = profissao
    }
}

pessoa = new Pai("João","Silva",40,"professor")
console.log(pessoa.nome + " tem " + pessoa.idade + " anos")
pessoa.fazAniverario()
console.log(pessoa.nome + " tem " + pessoa.idade + " anos")
```

```
[Running] node "c:\Users\Ferrara
João tem 40 anos
João tem 41 anos
```