Desenvolvimento com Javascript

Professor Rodrigo Bossini

Conteúdo

Capítulo 1

Introdução à linguagem Javascript

Neste capítulo tratamos dos aspectos fundamentais da linguagem Javascript.

1.1 Declaração de variáveis e constantes. Javascript é uma linguagem **dinamicamente tipada**. Isso quer dizer que o tipo de uma expressão é inferido em tempo de execução. Isso é diferente do que acontece com outras linguagens. Java, por exemplo, é uma linguagem estaticamente tipada. Isso quer dizer que o tipo das expressões é conhecido em tempo de compilação, o que permite que o compilador desempenhe diferentes tipos de validações. Em Javascript, há duas palavras reservadas para a declaração de variáveis: `let` e `var`. A palavra `const` é usada para a declaração de constantes. Veja o Bloco de Código 1.1.1.

Bloco de Código 1.1.1

```
1   //declarando constantes
2   const nome = "Jose";
3   const idade = 27;
4   // aspas simples e duplas têm o mesmo efeito
5   const sexo = "M";
6   const endereco = 'Rua K, 12'
7   //declarando variáveis
8   //let: variável local com escopo de bloco
9   let a = 2;
10  let b = "abc";
11  //var: seu escopo é a função em que foi declarada ou global
12  var c = 2 + 3;
13  var d = "abcd"
```

A palavra `let` foi introduzida na especificação **ES6**. É preferível utilizá-la pois o funcionamento de `var` pode ser contraintuitivo. O Bloco de Código 1.1.2 mostra alguns exemplos. A palavra `var` é mantida na linguagem apenas por retrocompatibilidade.

Bloco de Código 1.1.2

```
1   var linguagem = "Javascript";
2   console.log("Aprendendo " + linguagem);
3   //nome pode ser redeclarada
4   var linguagem = "Java";
5   console.log("Aprendendo, " + linguagem);
6
7   //escopo não restrito a bloco
8   var idade = 18;
9   //exibe undefined. Ou seja, a variável já existe aqui, só
        não teve valor atribuído.
10  //Ela é içada - do inglês hoist - para fora do bloco if
11  console.log(`Oi, ${nome}`);
12  if (idade >= 18) {
13    var nome = "João";
14    console.log(`Parabéns, ${nome}. Você pode dirigir`);
15  }
16  //ainda existe aqui
17  console.log(`Até mais, ${nome}.`);$
```

1.2 Tipos. Como mencionado anteriormente, Javascript é uma linguagem dinamicamente tipada. Veja os tipos existentes. É importante observar que valores primitivos são imutáveis. Objetos podem ser mutáveis ou imutáveis.

- Primitivos
  - boolean
  - null
  - number
  - string
  - undefined
- Objetos
  - JSON
  - Array
  - Classes Wrapper (String, Number, Boolean)
  - Date
  - Math
  - Funções

Professor Rodrigo Bossini
professorbossini@gmail.com

1.3 Coerção. Algumas linguagens de programação possuem um mecanismo conhecido como *coerção*, do inglês *cast*. Quando dois primitivos de tipos diferentes estão envolvidos em uma expressão, um deles pode ter seu tipo alterado[1] para que a expressão faça sentido. A coerção se refere a essa troca de tipo. Ela pode ocorrer de maneira explícita ou implícita. Veja o Bloco de Código 1.3.1.

Bloco de Código 1.3.1

```
1     const n1 = 2;
2     const n2 = '3';
3     //coerção implícita de n1, concatenação acontece
4     const n3 = n1 + n2;
5     console.log(n3);
6     //coeração explícita, soma acontece
7     const n4 = n1 + Number (n2)
8     console.log(n4)
```

1.4 Comparação. Javascript possui dois operadores de comparação.

- == A comparação leva em conta somente os valores envolvidos. Isso quer dizer que, caso sejam de tipos diferentes, ocorrerão coerções implícitas, as quais nem sempre têm o funcionamento mais intuitivo.
- === Este operador não realiza coerções. O resultado da comparação é *true* caso os valores e seus respectivos tipos forem iguais. Caso contrário, o resultado é *false*.

Veja alguns exemplos no Bloco de Código 1.4.1.

Bloco de Código 1.4.1

```
1   console.log(1 == 1)//true
2   console.log (1 == "1") //true
3   console.log (1 === 1) //true
4   console.log (1 === "1")//false
5   console.log (true == 1) //true
6   console.log (1 == [1])//true
7   console.log (null == null)//true
8   console.log (null == undefined)//true
9   console.log([] == false)//true
10  console.log ([] == [])//false
```

O Link 1.4.1 mostra uma tabela de comparação utilizando o operador ==. Seu funcionamento é mantido nas versões mais modernas de Javascript por razões como a retrocompatibilidade. Entretanto, é recomendável não utilizá-lo.

[1] Mais precisamente substituído por outro primitivo cujo tipo é o de interesse.

Professor Rodrigo Bossini
professorbossini@gmail.com

Link 1.4.1
https://dorey.github.io/JavaScript-Equality-Table/unified/

1.5 Vetores O Bloco de Código 1.5.1 mostra alguns exemplos de uso de vetores.

Bloco de Código 1.5.1

```
//declaração
v1 = [];
//podemos acessar qualquer posição, começando de zero
v1[0] = 3.4;
v1[10] = 2;
v1[2] = "abc"
//aqui, v1 tem comprimento igual a 11
console.log(v1.length)
//inicializando na declaração
v2 = [2, "abc", true]
console.log(v2)
//iterando
for (let i = 0; i < v2.length; i++){
  console.log(v2[i])
}
```

Em Javascript, vetores possuem diversos métodos utilitários. Veja os exemplos do Bloco de Código 1.5.2.

Bloco de Código 1.5.2

```
  const nomes = ["Ana Maria", "Antonio", "Rodrigo", "Alex",
      "Cristina"];
  const apenasComA = nomes.filter((n) => n.startsWith("A"));
  console.log(apenasComA);

  const res = nomes.map((nome) => nome.charAt(0));
  console.log(res);

  const todosComecamComA = nomes.every((n) =>
      n.startsWith("A"));
  console.log(todosComecamComA);

  const valores = [1, 2, 3, 4];
  const soma = valores.reduce((ac, v) => ac + v);
  console.log(soma);
```

1.6 Funções Javascript possui formas diferentes para se criar funções: blocos de código com nome - ou não - que podem ser colocados em execução em algum momento. A forma tradicional para se criar funções em Javascript envolve a palavra `function` Veja os exemplos do Bloco de Código 1.6.1.

Bloco de Código 1.6.1

```
function hello (){
  console.log ('Oi')
}
hello()
//cuidado, aqui redefinimos a função sem parâmetros
function hello (nome){
  console.log ('Hello, ' + nome)
}
hello('Pedro')

function soma (a, b) {
 return a + b;
}
const res = soma (2, 3)
console.log (res)
```

Também é possível criar funções anônimas. Uma vez criadas, elas podem ser atribuídas a variáveis ou constantes, como no Bloco de Código 1.6.2.

Bloco de Código 1.6.2

```
const dobro = function (n) {
    return n * 2;
};
const res = dobro(4);
console.log(res);
//valor padrão para o parâmetro
const triplo = function (n = 5) {
    return 3 * n;
};
console.log(triplo());
console.log(triplo(10));
```

A terceira possibilidade envolve o recurso chamado **arrow function**. Quando escrevemos uma arrow function, especificamos somente a sua lista de parâmetros e o seu corpo. Há um símbolo => - daí o nome arrow - entre eles. Uma arrow function não tem nome e também pode ser armazenada em constantes ou variáveis. Além disso, arrow functions têm as seguintes características.

- Quando a lista de parâmetros possui um único argumento, os parênteses

podem ser omitidos.

- Quando o corpo possui uma única instrução, as chaves podem ser omitidas.
- Quando o corpo possui uma única instrução que produz um valor a ser devolvido, a instrução `return` é opcional: Se usar as chaves, deve-se usar o return. Caso contrário, ele não pode ser usado.

Veja os exemplos do Bloco de Código 1.6.3.

Bloco de Código 1.6.3

```
1  const hello = () => console.log("Hello");
2  hello();
3  const dobro = (valor) => valor * 2;
4  console.log(dobro(10));
5  const triplo = (valor) => {
6    return valor * 3;
7  };
8  console.log(triplo(10));
9  //e agora?
10 const ehPar = (n) => {
11   n % 2 === 0;
12 };
13 console.log(ehPar(10));
```

1.7 Closures Para entender o que é um *closure*, é importante estudar entender alguns conceitos. Primeiro, em Javascript, funcões são **cidadãs de primeira classe**. Informalmente, um cidadão de primeira classe em uma linguagem de programação é uma entidade que oferece suporte a operações como as seguintes.

- Ser passada como argumento para uma função.
- Ser devolvida por uma função.
- Ser atribuída a uma variável.

O Bloco de Código 1.7.1 mostra como funções em Javascript podem estar envolvidas em todas as operações mencionadas. Há também o conceito de **função de alta ordem**. Uma função de alta ordem é aquela que recebe pelo menos uma função como parâmetro e/ou devolve uma função quando seu processamento termina.

Bloco de Código 1.7.1

```
/*uma função pode ser atribuída
a uma variável*/
let umaFuncao = function () {
  console.log ("Fui armazenada em uma variável");
}
//e pode ser chamada assim
umaFuncao()
/*f recebe uma função como parâmetro e, por isso
é uma função de alta ordem.
Por devolver uma função, g também é de alta ordem.
*/
function f (funcao) {
  //chamando a função
  //note como a tipagem dinâmica tem seu preço
  funcao()
}
function g () {
  function outraFuncao(){
    console.log("Fui criada por g");
  }
  return outraFuncao;
}
//f pode ser chamada assim
f (function (){
  console.log ('Estou sendo passada para f')
})
//e g pode ser chamada assim
const gResult = g()
gResult()
//e assim também
g()()
//outros testes
/* f chama g, que somente devolve uma função.
Nada é exibido.*/
f(g)
/*f chama a função devolvida por g.
"Fui criada por g" é exibido.*/
f(g())
/*f tenta chamar o que a função criada por g
devolve. Ela não devolve coisa alguma. Por isso,
um erro - somente em tempo de execução - acontece. */
f(g()())
//O que acontece?
f(1)
```

Uma função, quando definida por outra, é chamada **função interna** e tem dois escopos: o **escopo interno** e o **escopo externo**. Seu escopo interno é delimitado pelas chaves que definem seu corpo. Seu escopo externo é delimitado pelas chaves que definem o corpo da função que a define. Seu escopo externo é também chamado de escopo léxico. Uma função interna pode acessar as variáveis definidas em seu escopo externo. Veja o exemplo do Bloco de Código 1.7.2.

Bloco de Código 1.7.2

```
function f () {
  let nome = 'João';
  function g () {
    console.log (nome);
  }
  g()
}
f()
```

Os exemplos exibidos pelo Bloco de Código 1.7.3 funcionam, muito embora as funções *ola* e *saudacoesFactory* já tenham terminado a sua execução no momento em que as funções que produzem são chamadas, o que sugere que suas variáveis locais já não estão acessíveis.

Bloco de Código 1.7.3

```
1     function ola(){
2       let nome = 'João';
3       return function (){
4         console.log ('Olá, João');
5       }
6     }
7
8     let olaResult = ola();
9     /*perceba que aqui a função ola já terminou.
10    É de se esperar que a variável nome já não
11    possa ser acessada.*/
12    olaResult();
13
14    //também vale com parâmetros
15    function saudacoesFactory(saudacao, nome){
16      return function (){
17        console.log (saudacao + ', ' + nome);
18      }
19    }
20    let olaJoao = saudacoesFactory ('Olá', 'João');
21    let tchauJoao = saudacoesFactory('Tchau', 'João');
22    olaJoao();
23    tchauJoao();
```

Definição

Uma função interna em conjunto com as variáveis de seu escopo externo é o que chamamos de closure.

O funcionamento de funções envolvendo closures, em alguns casos, pode ser contra-intuitivo. Veja o Bloco de Código 1.7.4.

Professor Rodrigo Bossini
professorbossini@gmail.com

Bloco de Código 1.7.4

```
function eAgora(){
  let cont = 1;
  function f1 (){
    console.log (cont);
  }
  cont++;
  function f2 (){
    console.log (cont);
  }
  //JSON contendo as duas funções
  return {f1, f2}
}

let eAgoraResult = eAgora();

/* neste momento, a funcao eAgora já
executou por completo e a variável
cont já foi incrementada. Seu valor final
é mantido e, assim, ambas f1 e f2 exibirão 2.
*/
eAgoraResult.f1();
eAgoraResult.f2();

```

Capítulo 2

JSON - Javascript Object Notation

Neste capítulo tratamos da representação de dados utilizando **JSON - Javascript Object Notation**.

2.1 Intuição JSON é um formato para representação de dados independente de tecnologia. Nos dias atuais, é de longe o mais utilizado na troca de mensagens (feitas por requisições HTTP, por exemplo) entre sistemas computacionais. A ideia é representar dados como coleções de pares chave/valor. Veja alguns exemplos de representações de "coisas" do mundo real usando JSON.

2.1.1 Uma pessoa se chama João e tem 17 anos. Sua representação JSON é exibida no Bloco de Código 2.1.1.

Bloco de Código 2.1.1

```
let pessoa = {
  nome: "João",
  idade: 17,
}
//o acesso a propriedades pode ser feito com ponto
console.log("Me chamo " + pessoa.nome);
//e com [] também
console.log("Tenho " + pessoa["idade"] + " anos");
```

2.1.2 Uma pessoa se chama Maria, tem 21 anos e mora na rua B, número 121. Sua representação JSON é exibida no Bloco de Código 2.1.2.

Bloco de Código 2.1.2

```
let pessoaComEndereco = {
  nome: "Maria",
  idade: 21,
  endereco: {
    logradouro: "Rua B",
    numero: 121,
  },
};
console.log(
  `Sou ${pessoaComEndereco.nome},
   tenho ${pessoaComEndereco.idade} anos
    e moro na rua ${pessoaComEndereco.endereco["logradouro"]}
    número ${pessoaComEndereco["endereco"]["numero"]}`
);
```

2.1.3 Uma concessionária tem CNPJ e endereço. Ela possui alguns carros em estoque. Cada um deles tem marca, modelo e ana de fabricação. Sua representação JSON é exibida no Bloco de Código 2.1.3.

Professor Rodrigo Bossini
professorbossini@gmail.com

Bloco de Código 2.1.3

```
let concessionaria = {
  cnpj: "00011122210001-45",
  endereco: {
    logradouro: "Rua A",
    numero: 10,
    bairro: "Vila J",
  },
  veiculos: [
    {
      marca: "Ford",
      modelo: "Ecosport",
      anoDeFabricacao: 2018,
    },
    {
      marca: "Chevrolet",
      modelo: "Onix",
      anoDeFabricacao: 2020,
    },
    {
      marca: "Volkswagen",
      modelo: "Nivus",
      anoDeFabricacao: 2020,
    },
  ],
};
for (let veiculo of concessionaria.veiculos) {
  console.log(`Marca: ${veiculo.marca}`);
  console.log(`Modelo: ${veiculo.modelo}`);
  console.log(`Ano de Fabricação:
      ${veiculo.anoDeFabricacao}`);
}
```

2.1.4 Uma calculadora realiza as operações de soma e subtração . Nada impede que funções sejam armazenadas em objetos JSON. Veja o Bloco de Código 2.1.4.

Professor Rodrigo Bossini
professorbossini@gmail.com

Bloco de Código 2.1.4

```
let calculadora = {
    //pode ser arrow function
    soma: (a, b) => a + b,
    //e função comum também
    subtracao: function (a, b) {
    return a - b;
  },
};
console.log(`2 + 3 = ${calculadora.soma(2, 3)}`);
console.log(`2 - 3 = ${calculadora.subtracao(2, 3)}`);
```

Evidentemente, há uma especificação precisa que diz o que é um objeto JSON válido. Ela pode ser encontrada na página acessível por meio do Link 2.1.1. Visite essa página e estude os grafos sintáticos ali definidos.

Link 2.1.1
https://www.json.org/json-en.html

Professor Rodrigo Bossini
professorbossini@gmail.com

Capítulo 3

Execução Síncrona e Assíncrona

Neste capítulo trataremos do modelo de execução do Javascript.

3.1 Modelo Single Threaded Ambientes de execução Javascript são *Single Threaded*. Isso quer dizer que há um único fluxo de execução. Não há execução de código em paralelo. Como mostra o Bloco de Código 3.1.1, as instruções são executadas uma após a outra, na ordem em que foram definidas. Não há a possibidade de uma instrução i executar antes de outra instrução $j (\forall i > j)$.

Bloco de Código 3.1.1

```
1  console.log('Eu primeiro')
2  console.log("Agora eu")
3  console.log("Sempre vou ser a última...:(")
```

Este pode ser um funcionamento desejável, como mostra o Bloco de Código 3.1.2.

Bloco de Código 3.1.2

```
1  const a = 2 + 7
2  const b = 5
3  //só faz sentido se os valores a e b já estiverem disponíveis
4  console.log(a + b)
```

Entretanto, pode ser o caso de uma determinada instrução não depender de uma outra, anterior a ela, para poder executar corretamente. Isso pode ser um problema pois a instrução que a antecede pode ser demorada. Para ilustrar essa possibilidade, vamos usar uma função cuja execução demora uma quantidade de segundos. A instrução que vem depois de sua chamada não depende do resultado que ela produz. Veja o Bloco de Código 3.1.3.

Nota. Não se preocupe com o eventual *warning* sobre *memory leak*. A função `demorada` emprega uma técnica conhecida como **espera ocupada** apenas para simular um procedimento computacional demorado.

Bloco de Código 3.1.3

```
function demorada(){
    const atualMais2Segundos = new Date().getTime() + 2000
    //não esqueça do ;, única instrução no corpo do while
    while (new Date().getTime() <= atualMais2Segundos);
    const d = 8 + 4
    return d
}
const a = 2 + 3
const b = 5 + 9
const d = demorada()
/*
o valor de e não depende do valor devolvido
pela função demorada.
*/
const e = 2 + a + b
console.log(e)
```

Esse modelo de execução é conhecido como **síncrono** ou **bloqueante**. Ambientes Javascript (como um navegador ou o NodeJS) são responsáveis por ele. Podemos empregar diferentes técnicas para obter um outro tipo de execução conhecido como **assíncrono** ou **não bloqueante**. Uma forma bastante simples - e antiga, embora suficiente para ilustrar didaticamente o conceito - consiste no uso da função `setTimeout`. Ela recebe dois parâmetros: uma função e um valor em milissegundos. A execução da função somente ocorre uma vez que pelo menos a quantidade de milissegundos especificada se esgote. Enquanto isso, as instruções que vem depois da chamada à função continuam executando normalmente, sem ficar esperando. Elas não ficam **bloqueadas**. Daí o nome do modelo. Veja um exemplo no Bloco de Código 3.1.4.

Professor Rodrigo Bossini
professorbossini@gmail.com

Bloco de Código 3.1.4

```
function demorada(){
  const atualMais2Segundos = new Date().getTime() + 2000
    //não esqueça do ;, única instrução no corpo do while
    while (new Date().getTime() <= atualMais2Segundos);
    const d = 8 + 4
    return d
}
const a = 2 + 3
const b = 5 + 9
//função será executada depois de, pelo menos, 500
    milissegundos
setTimeout(function(){
    const d = demorada()
    console.log(d)
}, 500)

//enquanto isso, essas linhas prosseguem executando
//sem ficar esperando
const e = a + b
console.log(e)
```

Embora o Bloco de Código 3.1.4 ilustre o processamento não bloqueante, é importante observar uma característica importante. A função que foi entregue como parâmetro à função `setTimeout` foi, na verdade, **enfileirada**. Ela somente vai executar depois de o bloco principal ter sido completamente executado. Veja o exemplo do Bloco de Código 3.1.5. Note que especificamos 0 no segundo argumento. Tecnicamente, ela poderia executar imediatamente. Porém, isso não acontecerá, devido ao enfileiramento.

Bloco de Código 3.1.5

```
setTimeout(function(){
console.log('dentro da timeout', 0)
})
const a = new Date().getTime() + 1000
//não esqueça do ;, única instrução no corpo do while
while (new Date().getTime() <= a);
console.log('fora da timeout')
```

Veja também o exemplo do Bloco de Código 3.1.6. Ele ilustra como o enfileiramento somente acontece depois de o tempo especificado no segundo parâmetro da função `setTimeout` se esgotar.

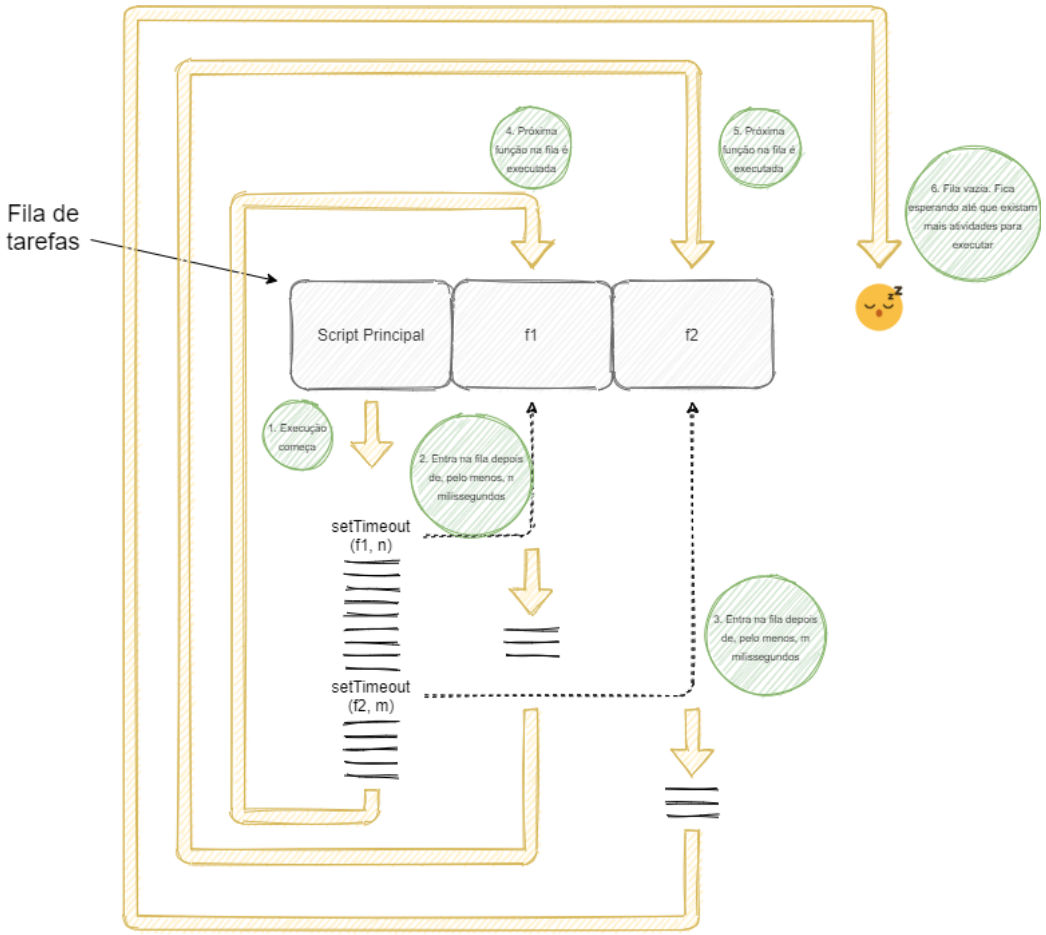
Professor Rodrigo Bossini
professorbossini@gmail.com

Bloco de Código 3.1.6

```
function demorada(tempo) {
  console.log(`demorada ${tempo}`);
  const atualMaisTempo = new Date().getTime() + tempo;
  //não esqueça do ;, única instrução no corpo do while
  while (new Date().getTime() <= atualMaisTempo);
  const d = 8 + 4;
  return d;
}
setTimeout(function (){demorada(2000)}, 2000);
setTimeout(function (){demorada(1000)}, 1000);
console.log("chegou ao fim do script principal");
```

A Figura 3.1.1 ilustra a estrutura denominada **event loop** - algo como laço de evento em português - existente em ambientes de execução Javascript.

Figura 3.1.1



Concluímos, assim, que todo o código Javascript que escrevemos executa em

Professor Rodrigo Bossini
professorbossini@gmail.com

uma única thread. Entretanto, é importante observar que há, de fato, instruções que executam em threads diferentes. Essas são gerenciadas pelo próprio ambiente de execução Javascript (NodeJs, navegador etc). Dizemos que o modelo é Single Threaded pois o desenvolvedor tem acesso somente a uma thread. Ele não escreve código para criar e gerenciar outras threads explicitamente. Isso fica a cargo do ambiente. No exemplo do Bloco de Código 3.1.7 usamos um módulo para acesso ao sistema de arquivos. Fazemos a leitura do conteúdo de um arquivo. Quando ela termina, o conteúdo é exibido. Todo o código que escrevemos executa em uma única thread. Entretanto, a leitura do arquivo, realizada pela função `readFile` *pode* executar em uma thread separada.

Bloco de Código 3.1.7

```
1  const fs = require("fs");
2  const abrirArquivo = function (nomeArquivo) {
3    const exibirConteudo = function (erro, conteudo) {
4      if (erro) {
5        console.log(`Deu erro: ${erro}`);
6      } else {
7        console.log(conteudo.toString());
8      }
9    };
10   fs.readFile(nomeArquivo, exibirConteudo);
11 };
12 //crie um arquivo chamado arquivo.txt com o conteúdo
       ``2'' (sem as aspas)
13 //no mesmo diretório em que se encontra seu script
14 abrirArquivo("arquivo.txt");
```

3.2 O inferno de callbacks As funções que entregamos como argumento para a função `setTimeout` e a função `exibirConteudo` usada no Bloco de Código 3.1.7 são exemplos de funções **callback**. A definição de uma função callback é responsabilidade do desenvolvedor. Colocá-la em execução, por outro lado, é responsabilidade do ambiente Javascript. Uma função callback entra em execução quando um evento determinado acontece. Há um fenômeno conhecido como **callback hell** ou **inferno de callbacks** que consiste no aninhamento de funções callback. Veja um exemplo no Bloco de Código 3.2.1. Desejamos dobrar o valor lido do arquivo *arquivo.txt* e armazenar o valor obtido em um arquivo chamado *dobro.txt*.

Bloco de Código 3.2.1

```
const fs = require("fs");
const abrirArquivo = function (nomeArquivo) {
  const exibirConteudo = function (erro, conteudo) {
    if (erro) {
      console.log(`Deu erro: ${erro}`);
    } else {
      console.log(conteudo.toString());
      const dobro = +conteudo.toString() * 2;
      const finalizar = function (erro){
          if(erro){
              console.log('Deu erro tentando salvar o dobro')
          }
          else{
              console.log("Salvou o dobro com sucesso");
          }
      }
      fs.writeFile('dobro.txt', dobro.toString(), finalizar);
    }
  };

  fs.readFile(nomeArquivo, exibirConteudo);
};
abrirArquivo("arquivo.txt");
```

O aninhamento de funções callback compromete a legibilidade do código. Daí o singelo nome **inferno de callbacks**. Há, ainda, outras características indesejáveis inerentes ao uso de callbacks.

- A ordem dos parâmetros de uma função callback varia. A função `exibirConteudo` do Bloco de Código 3.2.1, por exemplo, recebe um objeto com dados referentes a um possível erro e um objeto com os dados caso a execução ocorra com sucesso, **nesta ordem**. Outras funções callback podem ser chamadas com a ordem invertida. Não há garantia. É sempre necessário verificar a documentação antes.

3.3 Promises Desde a especificação **ECMAScript** 2015, a linguagem Javascript conta com um recurso chamado **Promise**. Trata-se de um mecanismo próprio para a manipulação de código assíncrono que visa simplificar as características inerentes ao uso de callbacks. A sua especificação oficial pode ser vista no Link 3.3.1.

Link 3.3.1
https://tc39.es/ecma262/#sec-promise-objects

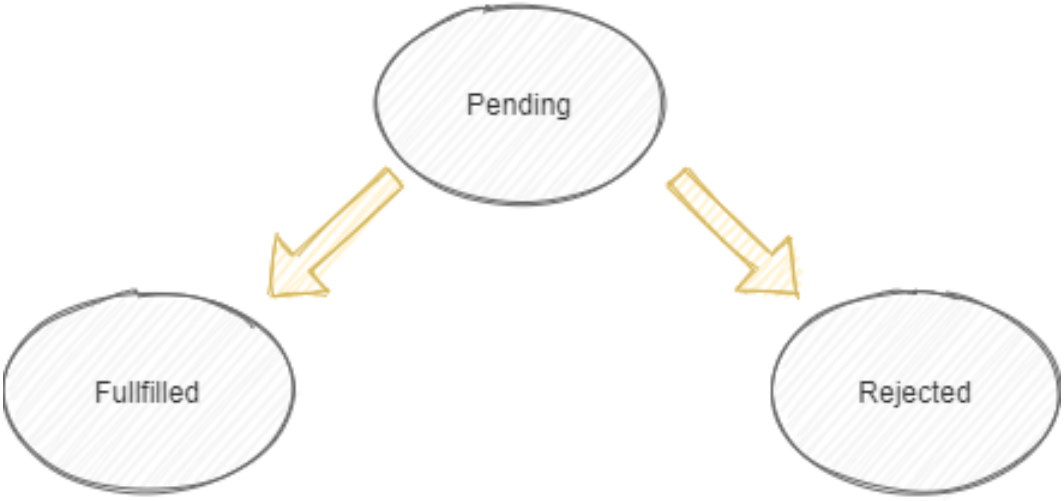
Nota. O uso de promises não implica na execução de código em paralelo. A ideia é simplificar a manipulação de código cuja execução se dá de maneira assíncrona. Como mostra a Figura 3.1.1, é possível que código assíncrono execute em uma única thread. Promises podem ser usadas tanto neste contexto quanto em outros em que exista a execução de código em paralelo gerenciada pelo ambiente Javascript.

Definição

Uma **Promise** é um objeto por meio do qual uma função pode propagar um resultado ou um erro em algum momento no futuro.

Como mostra a Figura 3.3.1, uma promise pode estar em um de três estados.

Figura 3.3.1

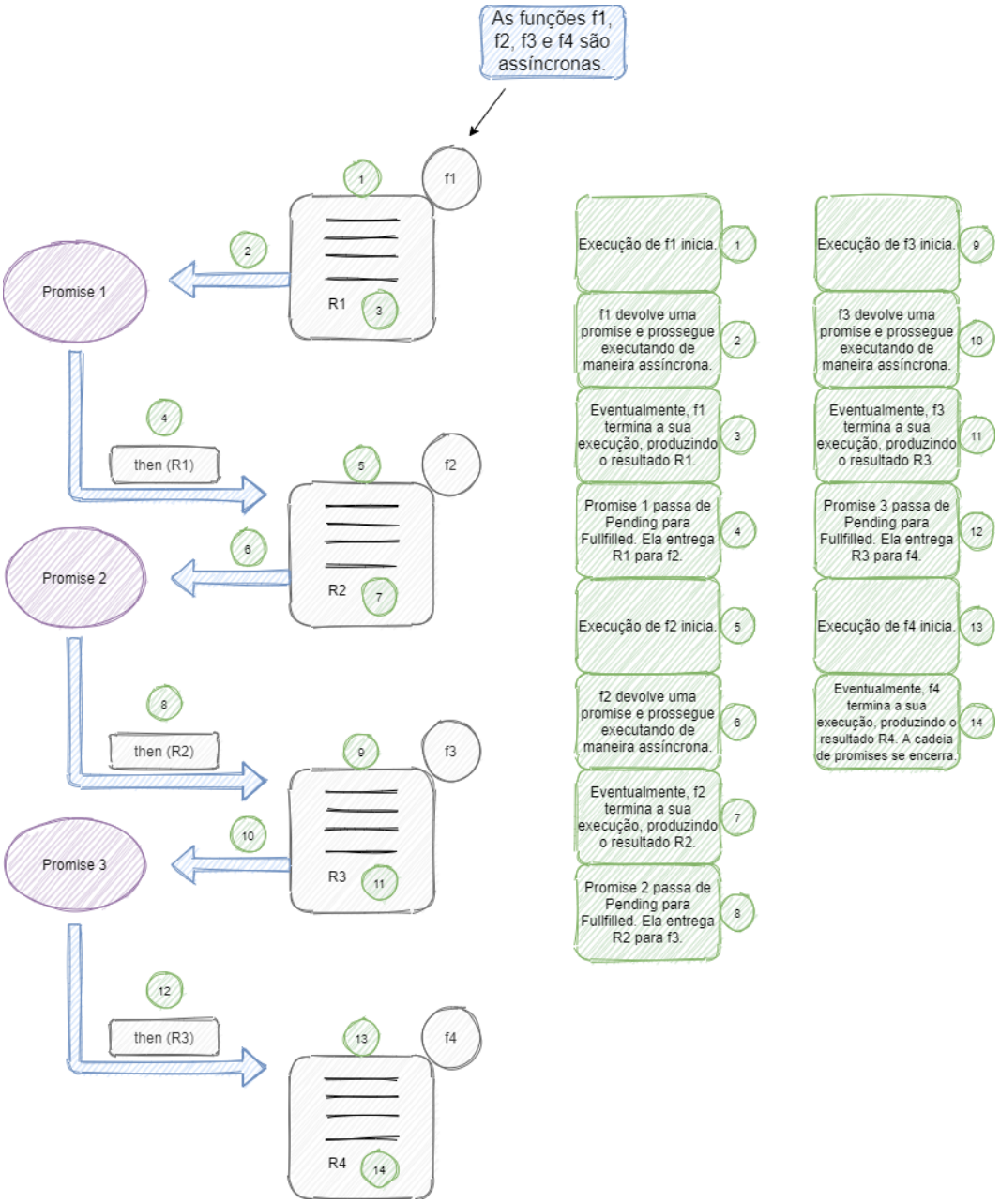


Seu significado e transição são os seguintes.

- Quando uma promise é produzida e o processamento associado a ela ainda não está concluído, ela está no estado **Pending**.
- Quando o processamento associado a uma promise termina com sucesso, ela passa para o estado **Fullfilled**.
- Quano o processamento associado a uma promise termina com erro, ela passa para o estado **Rejected**.
- Os estados **Fullfilled** e **Rejected** são **estados finais**. Uma vez que uma promise se encontre em um desses estados, ela nunca transita para outro estado.
- Uma promise pode ser criada em qualquer um dos três estados.

Uma das vantagens obtidas pelo uso de promises é a simplificação da passagem de parâmetros entre funções assíncronas. Como mostra a Figura 3.3.2, a sua execução pode ser **encadeada**.

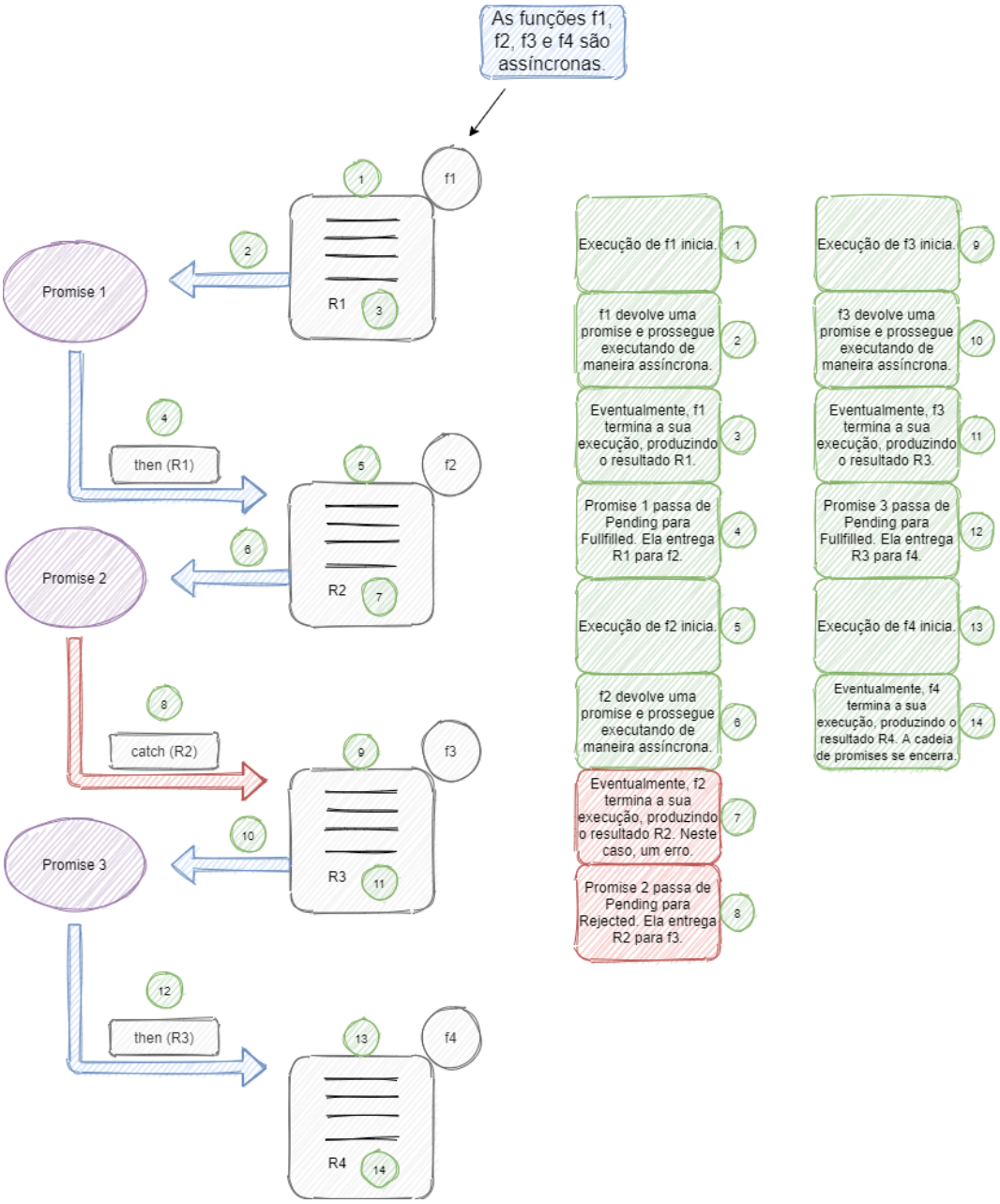
Figura 3.3.2



A Figura 3.3.2 ilustra um fluxo em que todas as promises envolvidas terminam com sucesso, passando do estado **Pending** para o estado **Fullfilled**. O encadeamento, neste caso, é feito por meio da função `then`. Também pode ser o caso de alguma delas terminarem com erro. Neste caso, o encadeamento é feito por meio da função `catch`. Ao longo de um encadeamento, as funções `then` e `catch` podem ser mescladas. Veja a Figura 3.3.3.

Professor Rodrigo Bossini
professorbossini@gmail.com

Figura 3.3.3



O uso das funções `then` e `catch` resolve um dos problemas inerentes ao uso de callbacks: não é necessário verificar a documentação de cada biblioteca utilizada para descobrir qual dos dois é entregue como primeiro argumento. O tratamento de resultados sempre se dá na função `then` e o tratamento de erros sempre se dá na função `catch`.

3.3.1 Construindo promises Uma função cuja execução tem potencial para demorar, idealmente executa de maneira assíncrona. Ela constrói um objeto do

tipo `Promise` e o devolve imediatamente, no estado **Pending**. A seguir, prossegue com a sua computação. Ela pode terminar com sucesso ou com erro. Caso termine com sucesso, a função especificada pelo cliente no bloco `then` entra em execução. Caso contrário, aquela especificada no bloco `catch` executa. No Bloco de Código 3.3.1, a função assíncrona devolve uma promise em estado **Pending**. Quando termina, ela chama a função `resolve`, o que quer dizer que a promise passou de **Pending** para **Fullfilled**.

Bloco de Código 3.3.1

```
1   function calculoDemorado(numero) {
2     return new Promise(function (resolve, reject) {
3       let res = 0;
4       for (let i = 1; i <= numero; i++) {
5         res += i;
6       }
7       resolve(res);
8     });
9   }
10  calculoDemorado(10).then( (resultado) => {
11      console.log(resultado)
12  })
```

Uma função assíncrona pode também devolver uma promise cujo estado é **Fullfilled**. Isso pode acontecer quando ela detectar que a resposta para o problema que pretende resolver é imediata. O somatório realizado no Bloco de Código 3.3.1 utiliza força bruta. O resultado pode ser obtido por meio de uma fórmula fechada, como mostra a Equação 3.3.1

$$\sum_{i=0}^{n} i = \frac{n(n+1)}{2} \forall n \geq 0 \tag{3.3.1}$$

Há quem diga que **Gauss** foi o responsável por obter esse resultado. Há também quem diga que ele é conhecido desde a Grécia antiga. Veja o Link 3.3.2.

Link 3.3.2
https://hsm.stackexchange.com/questions/384/did-gauss-find-the-formula-for-123-ldotsn-2n-1n-in-elementary-school

Dado que o cálculo não precisa ser demorado, a função assíncrona pode devolver uma promise já no estado **Fullfilled**, como no Bloco de Código 3.3.2.

Bloco de Código 3.3.2

```
function calculoRapidinho (numero){
    return Promise.resolve((numero * (numero + 1)) / 2);
}
calculoRapidinho (10).then(resultado =>{
    console.log (resultado)
})
//Executa primeiro, mesmo que a promise já esteja fullfilled
console.log('Esperando...')
```

Um promise também pode ser devolvida já no estado **Rejected**. Para este exemplo, pode ser interessante fazê-lo caso o valor entregue para a função assíncrona seja negativo, como no Bloco de Código 3.3.3. Note que o código cliente pode especificar funções para ambas as possibilidades. Somente uma delas executará.

Bloco de Código 3.3.3

```
function calculoRapidinho(numero) {
  return numero >= 0
    ? Promise.resolve((numero * (numero + 1)) / 2)
    : Promise.reject("Somente valores positivos, por favor");
}

calculoRapidinho(10)
  .then((resultado) => {
    console.log(resultado);
  })
  .catch((err) => {
    console.log(err);
  });
calculoRapidinho(-1)
  .then((resultado) => {
    console.log(resultado);
  })
  .catch((err) => {
    console.log(err);
  });
console.log("esperando...");
```

Há diversos serviços disponíveis no portal **OpenWeatherMap**, acessível por meio do Link 3.3.3, que permitem realizar consultas referentes a previsões do tempo.

Link 3.3.3
https://openweathermap.org/

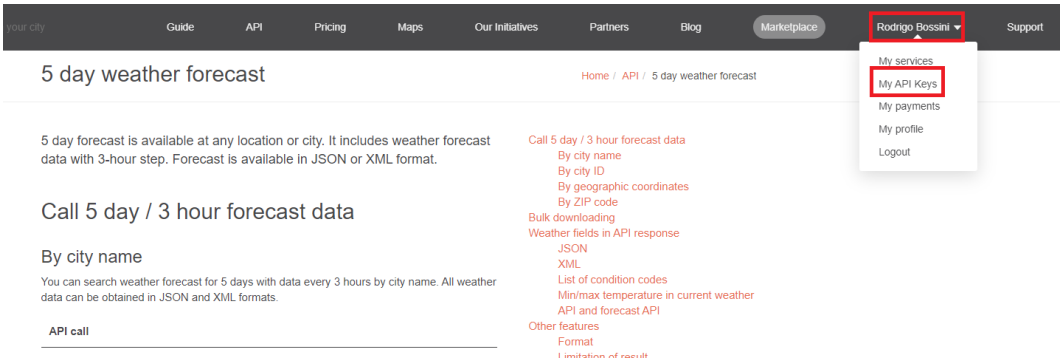
Professor Rodrigo Bossini
professorbossini@gmail.com

Dentre aqueles que permitem acesso gratuito, encontra-se o serviço **5 Day / 3 Hour Forecast**. Ele entrega previsões do tempo para até os próximos cinco dias, de três em três horas. Veja a sua documentação no Link 3.3.4.

Link 3.3.4
https://openweathermap.org/forecast5

A seguir, utilizaremos este serviço para ilustrar o uso do encadeamento de promises. Antes de mais nada, crie uma conta para você no portal. Uma vez que tenha criado a conta - é possível que receba um email para confirmá-la - e feito login, clique no canto superior direito em *seu nome » My API Keys* como mostra a Figura 3.3.4.

Figura 3.3.4



Na tela seguinte, gere uma chave de API para você. Escolha um nome que achar apropriado. Para realizar as requisições, utilizaremos o pacote **axios**. Ele pode ser instalado com

```
npm install axios
```

Feita a instalação, importe o pacote e declare as constantes ilustradas no Bloco de Código 3.3.4.

Professor Rodrigo Bossini
professorbossini@gmail.com

Bloco de Código 3.3.4

```
1   const axios = require("axios");
2   //sua chave aqui
3   const appid = "sua_chave_aqui";
4   //cidade desejada
5   const q = "Itu";
6   //unidade de medida de temperatura
7   const units = "metric";
8   //idioma
9   const lang = "pt_BR";
10  //quantidade de resultados
11  const cnt = "10"
12  const url = `https://api.openweathermap.org/data/2.5/fore
        cast?q=${q}&units=${units}&appid=${appid}&lang=${lang
        }&cnt=${cnt}`;$
```

O Bloco de Código 3.3.5 mostra como fazer uma requisição e encadear diversas promises. Cada função especificada resolve um problema possivelmente de interesse. Quando termina, repassa o resultado por meio de uma nova promise.

Bloco de Código 3.3.5

```
//faz a requisição
axios
  .get(url)
  .then((res) => {
    //mostra o resultado e devolve somente a parte de
        interesse
    console.log(res);
    return res.data;
  })
  .then((res) => {
    //mostra o total e devolve o resultado
    console.log(res.cnt);
    return res;
  })
  .then((res) => {
    //devolve somente a lista de previsões
    console.log("aqui", res);
    return res["list"];
  })
  .then((res) => {
    //para cada resultado, mostra algumas informações
    for (let previsao of res) {
      console.log(`
            ${new Date(+previsao.dt * 1000).toLocaleString()},
            ${'Min: ' + previsao.main.temp_min}\u00B0C,
            ${'Max: ' + previsao.main.temp_max}\u00B0C,
            ${'Hum: ' + previsao.main.humidity}%,
            ${previsao.weather[0].description}
        `);
    }
    return res;
  })
  .then((res) => {
      //verifica quantas previsões têm percepção humana
          de tempertura acima de 30 graus
      const lista = res.filter(r => r.main.feels_like >= 30);
      console.log (`${lista.length} previsões têm
          percepção humana de temperatura acima de 30
          graus`)
  });
```


Professor Rodrigo Bossini
professorbossini@gmail.com

3.3.2 Async/await É verdade que o uso de promises é vantajoso quando comparado ao uso de callbacks. Entretanto, o encadeamento de promises usando `then` e `catch` é significaticamente mais complexo do que a execução sequencial bloqueante. A especificação **ECMAScript 2017** inclui um recurso que permite a execução de funções assíncronas envolvendo promises sem ter de lidar diretamente com as funções `then` e `catch`. Este recurso é caracterizado pelas palavras-chave `async` e `await`. A palavra `async` pode preceder o nome de uma função, no momento em que ela é definida. Os efeitos são os seguintes.

- A função executa de forma assíncrona. Caso em sua definição original ela devolva um valor qualquer, uma vez que tenha sido marcada com `async`, ela passa a devolver uma promise que permite a obtenção daquele valor.
- Uma chamada de função assíncrona feita por ela pode ser precedida pela palavra `await`. Neste caso, a função chamada deixará de retornar uma promise imediatamente. Ela irá prosseguir com seu processamento e somente devolver o resultado quando estiver pronto. Ela executa, portanto, como se fosse síncrona.

O Bloco de Código 3.3.6 mostra um exemplo em que uma função que originalmente executa de maneira síncrona é marcada com a palavra `async`. Ela passa a devolver uma promise que permite a obtenção do resultado. O código cliente pode, portanto, aplicar as funções `then` e `catch`.

Bloco de Código 3.3.6

```
1  async function hello(nome) {
2      return "Oi, " + nome;
3  }
4  const boasVindas = hello("João");
5  console.log(boasVindas);
6  boasVindas.then((res) => console.log(res));
```

Para ilustrar o uso da palavra `await`, vamos utilizar uma função assíncrona que calcula o fatorial de um número inteiro recebido como parâmetro. Ela toma o cuidado de verificar se o valor passado é negativo. Veja o Bloco de Código 3.3.7.

Bloco de Código 3.3.7

```
1  function fatorial(n) {
2    if (n < 0) return Promise.reject("Valor não pode ser
         negativo");
3    let res = 1;
4    for (let i = 2; i <= n; i++) res *= i;
5    return Promise.resolve(res);
6  }
```

Como vimos, ela pode ser chamada e ter seu resultado tratado com as funções `then` e `catch`. Veja o Bloco de Código 3.3.8.

Bloco de Código 3.3.8

```
function chamadaComThenCatch() {
  fatorial(5)
    .then((res) => console.log(res))
    .catch((res) => console.log(res));

  fatorial(-1)
    .then((res) => console.log(res))
    .catch((res) => console.log(res));
}
chamadaComThenCatch();
```

Usando a palavra `await`, podemos fazer chamadas mais simples, sem utilizar `then` e `catch`. Veja o Bloco de Código 3.3.9.

Bloco de Código 3.3.9

```
//para usar await tem que ser async
async function chamadaComAwait() {
  //note que não há paralelismo implícito
  //somente haverá paralelismo se a função chamada
      utilizar explicitamente
  const f1 = await fatorial(5);
  console.log(f1);
  const f2 = await fatorial(-1);
  console.log(f2);
}
```

Assim, a palavra `async` pode ser usada para tornar uma função síncrona em uma função assíncrona. As palavras `async` e `await` podem ser utilizadas em conjunto para simplificar o uso de promises, descartando o uso de `then` e `catch`.

.

Professor Rodrigo Bossini
professorbossini@gmail.com