

PROMISE

AULA 11

Promise (Promessa). É uma função. Uma função que pode ser cumprida, ou seja, ser executada sem erros, ou não, quando ocorre algum erro dentro dessa função ou algo não planejado.

A *promise* sempre é passada como parâmetro PARA UMA OUTRA função.

Uma *promise* sempre tem sempre dois parâmetros: *resolve* e *reject*. E através desses parâmetros que sabemos se a função *promise* foi executada com sucesso ou se ocorreu algum erro dentro dela. Quando ela foi executada com sucesso, colocamos valor no parâmetro *resolve*. E quando ocorreu algum erro ou algo que não era esperado, colocamos valor no parâmetro *reject*.

OBS: Tanto o *resolve* quanto o *reject* recebem apenas um único valor.

Quando chamamos uma função que usa uma *promisse*, podemos usar o método **.then** ou o método **.catch**.

De forma que o código presente no método **.then** será executado quando a *promise* for executada sem qualquer erro, ou seja, quando a *promise* tiver entrado no parâmetro *resolve*.

Ao passo que o **.catch** será executado quando ocorreu alguma falha na *promise* e o parâmetro *reject* acabou recebendo valor. Veja na figura ao lado.

É importante destacar que tanto o método **.then** quanto o método **.catch** “estão de posse” de uma informação. Que será o o valor do parâmetro que foi retornado na *promise*. Ou seja, o valor que foi atribuído ao *resolve* ou o valor que foi atribuído ao *reject*.

E para obter essa “informação” usamos uma variável como parâmetro. No exemplo ao lado, usamos a variável *mensagemRetornoResolve*, no **.then**, e a variável *mensagemRetornoReject*, no **.catch**.

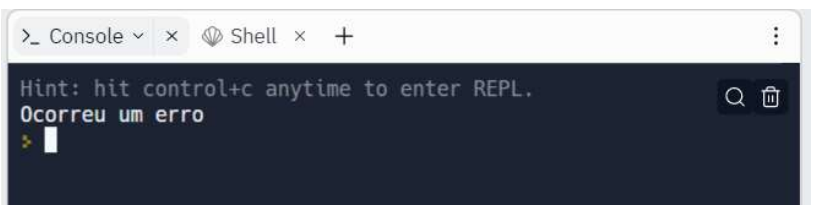
```
2 function cadastrarUsuario()
3 {
4     //Criando a promise!
5     return new Promise((resolve, reject) => {
6
7         var erro = true;
8
9         if(!erro){
10             resolve('Sucesso!');
11         }else{
12             reject('Ocorreu um erro');
13         }
14     });
15 }
16
```

```
2 function cadastrarUsuario()
3 {
4     //Criando a promise!
5     return new Promise((resolve, reject) => {
6
7         var erro = true;
8
9         if(!erro){
10             resolve('Sucesso!');
11         }else{
12             reject('Ocorreu um erro');
13         }
14     });
15 }
16
17 //chamando a função que tem uma promise.
18 cadastrarUsuario()
19
20
21 /*vai executar o código abaixo APENAS se a promise foi executada com sucesso,
22 ou seja, se o RESOLVE recebeu valor.*/
23 .then((mensagemRetornoResolve) => {
24     console.log(mensagemRetornoResolve);
25 })
26
27 /*vai executar o código abaixo APENAS se a promise foi executada com FALHA,
28 ou seja, se o REJECT recebeu valor.*/
29 .catch((mensagemRetornoReject) => {
30     console.log(mensagemRetornoReject);
31 })
32
```

Resultado ao executar o programa acima:

Disponível em:

<https://replit.com/join/lsnwmkotiw-fabiosiqueira1>




AULA 12

Na aula 11 foi mostrado que o `.catch` e o `.then` são uma forma de verificarmos o retorno de uma função *promise*. Ou seja, se ela foi executada com sucesso (*resolve*) ou se ocorreu alguma falha (*reject*).

É MUITO IMPORTANTE SABER QUE tanto o `.catch` quanto o `.then` SÃO EXECUTADOS SOMENTE QUANDO A PROMISSE É FINALIZADA. Isso pode parecer obvio, pelo que vimos até agora, mas a partir deste módulo começamos a ver programação assíncrona. Isso significa que o programa estará EXECUTANDO AO MESMO TEMPO várias instruções (funções ou comandos). Isso é novidade!

Um exemplo prático de programação assíncrona pode ser visto se colocarmos um comando `console.log` ao término do programa feito anteriormente. Repare na figura abaixo que, mesmo o `console.log` sendo colocado no final do programa, sua mensagem é exibida primeiro que o retorno da função `cadastrarUsuario`. Isso ocorre porque a função `cadastrarUsuario` chama uma *promise*. E a *promise* é uma função assíncrona. Isso faz com que o Repl.it execute o comando seguinte (`console.log("oi")`) mesmo sem ter terminado de executar a função “`cadastrarUsuario`”.



```
index.js > f cadastrarUsuario > ...
7   var erro = true;
8
9   if(erro == false){
10     resolve('Sucesso!');
11   }else{
12     reject('Ocorreu um erro');
13   }
14
15 };
16 }
17
18 //chamando a função que tem uma promise.
19 cadastrarUsuario()
20
21 /*vai executar o código abaixo APENAS se a promise foi executada com sucesso,
22 ou seja, se o RESOLVE recebeu valor.*/
23 .then((mensagemRetornoResolve) => {
24   console.log(mensagemRetornoResolve);
25 })
26
27 /*vai executar o código abaixo APENAS se a promise foi executada com FALHA,
28 ou seja, se o REJECT recebeu valor.*/
29 .catch((mensagemRetornoReject) => {
30   console.log(mensagemRetornoReject);
31 });
32
33 console.log("oi");
34
```

oi
Hint: hit control+c any
Ocorreu um erro

Mesmo sendo executado depois, o resultado foi mostrado primeiro!

Após relembrado que estamos lidando com funções assíncronas... na aula 12, é mostrada uma outra forma de se chamar uma *promise*. Nesta nova forma, não se usa os métodos `.catch` e `.then`.

Ao invés disso, podemos “jogar” para uma variável o que será retornado da função “`cadastrarUsuario`”, mais ou menos assim, segundo o exemplo da página anterior.

```
var dados = cadastrarUsuario();
```

Porém, por ser uma função assíncrona, o programa continua executando as outras instruções ao mesmo tempo que executa a função `cadastrarUsuario`. Isso fará com que o programa tente jogar o valor para a variável “dados” antes da função `cadastrarUsuario` ter sido concluída. Ou seja, antes da `promise` ter retornado seu valor. Por isso a mensagem “pendente” é exibida, conforme mostrado no minuto 3:00 da videoaula 12.

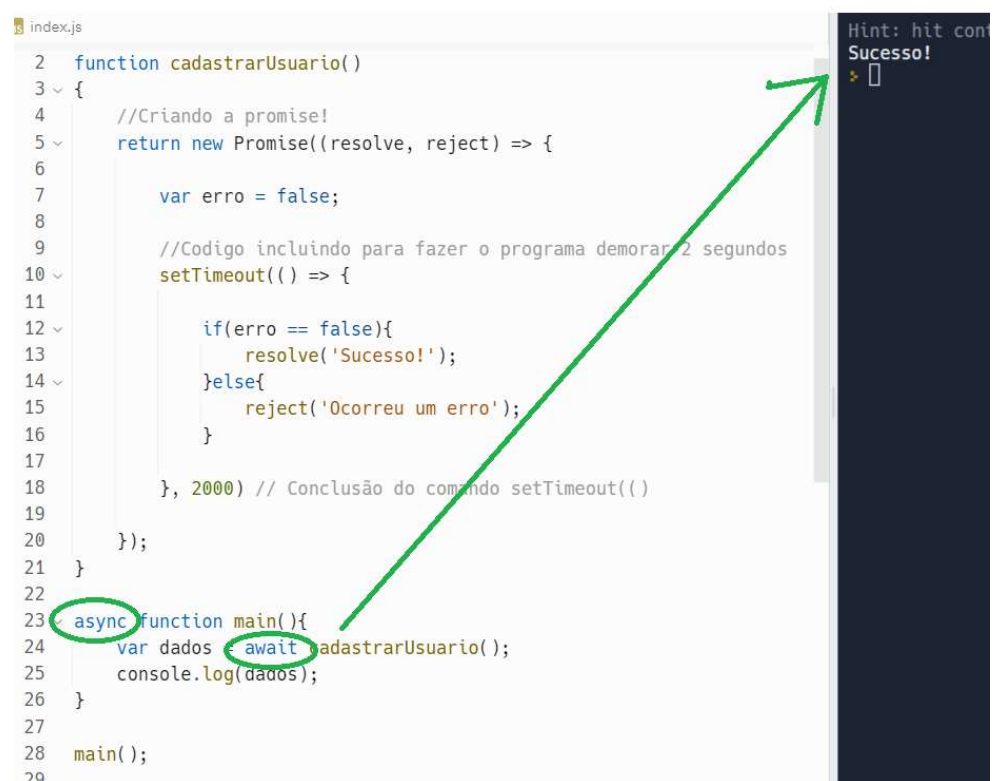


```
index.js
1
2 function cadastrarUsuario()
3 {
4   //Criando a promise!
5   return new Promise((resolve, reject) => {
6
7     var erro = false;
8
9     //Codigo incluindo para fazer o programa demorar 2 segundos
10    setTimeout(() => {
11
12      if(erro == false){
13        resolve('Sucesso!');
14      }else{
15        reject('Ocorreu um erro');
16      }
17
18    }, 2000) // Conclusão do comando setTimeout()
19  });
20 }
21
22
23 var dados = cadastrarUsuario();
24 console.log(dados);
```

Promise { <pending> }
Hint: hit control+c anytime

Para resolver esse “problema” existe os comandos **await** e **async**. Sendo que:

- **await**: deve ser usado antes da chamada da função. Ex:
`var dados = await cadastrarUsuario();`
Isso fará com que o programa “ESPERE” o término da execução da função `cadastrarUsuario` para prosseguir com a execução das demais instruções.
- **async**: Conforme mostrado na videoaula 12, a função **await** deve ser usada apenas dentro de outras funções e essa “outra função” deve ter sido classificada como assíncrona, usando o comando **async**. Veja figura ao lado.



```
index.js
2 function cadastrarUsuario()
3 {
4   //Criando a promise!
5   return new Promise((resolve, reject) => {
6
7     var erro = false;
8
9     //Codigo incluindo para fazer o programa demorar 2 segundos
10    setTimeout(() => {
11
12      if(erro == false){
13        resolve('Sucesso!');
14      }else{
15        reject('Ocorreu um erro');
16      }
17
18    }, 2000) // Conclusão do comando setTimeout()
19  });
20 }
21
22
23 async function main(){
24   var dados = await cadastrarUsuario();
25   console.log(dados);
26 }
27
28 main();
29
```

Hint: hit control+c anytime
Sucesso!