```
//PROMISE:

//As promises são um tipo de função do Javascript que utilizamos para basear as nossas requisições de um conteúdo por através de URL na rede. Por que são chamadas de promises? Por que esse é justamente o conceito de uma requisição URL, quando fazemos uma requisição, temos a promessa de que o recurso estará realmente disponível ou se infelizmente ele não estará, gerando um erro na nossa aplicação.

//As promises foram criadas justamente para lidar com essas requisições, onde podemos adicionar um tratamento ao conteúdo informático caso a promessa seja resolvida, ou poderemos dar um tratamento de erro caso a promessa seja rejeitada. Trabalhando nesses 2 conceitos, a promisse possuí 2 métodos, um para resolução e outro para rejeição:
//      * .then() - usado quando uma promise recebeu o dado informático;
//      * .catch() - usado quando ocorreu algum tipo de erro e a promise não conseguiu o conteúdo informático;

//Tanto ".then()" quanto ".catch()" recebem callbacks que irão realizar um tratamento sobre os dados de acordo com as suas respectivas funcionalidades, o que significa que, "then()" só vai receber dados quando o promise tiver dado certo, e "catch()" só vai receber dados quando o promise tiver dado errado. Como then() e catch() sabem que os dados deram errado ou deram certo? Isso acontece por que promise recebe 2 parâmetros:
/*      promise(resolve, reject)
            onde "resolve" será uma callback que será executada se o promise tiver dado certo e "reject" é outra callback
            que será executada quando promisse tiver dado errado...*/

//Um detalhe importante das promises é que elas sempre irão trabalhar com chamadas assícronas, ou seja, enquanto o código executa, a promessa é chamada e procura por uma resolução retornando ela quando o processo acaba.

//Promise, faz parte de uma função construtora, por isso devemos instanciar um objeto onde desejamos gerar um promise...

//GERANDO UM PROMISE FICTÍCIO:
//Abaixo estamos gerando um promise que recebe uma requisição e uma resposta positiva, note que esse promisse possuí um time para acontecer, mostrando que ele é assícrono e só é executado quando o time acaba, depois temos uma execução independente que será executada antes do promise...
function falarDepoisDe(segundos, frase) {
    return new Promise((resolve, reject) => { //Veja que o retorno da função irá retornar uma instância de um promise que recebe 2 parâmetros, o 1º é sempre uma função callback que será executada se o promisse tiver dado certo e o 2º sempre será uma função callback para ser executada se o promise tiver dado errado...
        setTimeout(() => { //Perceba que dentro do promisse temos uma função arrow que vai executar o resolve - caso os dados sejam pegos - ou o reject - caso ocorra algum erro.
```

```
            resolve(frase) //Note que tanto resolve quanto reject só podem receber um parâmetro, se quisermos que mais parâmetros sejam passados temos que usar um objeto...
            reject(frase)
        }, segundos * 1000)
    })
}

falarDepoisDe(2, 'Essa frase veio de um promise...') //Aqui temos a passagem de parâmetros para a função que irá gerar o promise, do jeito que está a execução irá entrar só no resolve, mas se desejar que ela gere um erro, apague um dos pâmetros...
    .then((frase) => frase.concat(' Sim ela veio')) //Veja que estamos usando then() e catch() encadeados, e assim como os callback de promisse, then() e catch() só podem receber 1 parâmetro...
    .then(fraseNova => console.log(fraseNova + ', mas só depois de 2 segundos'))
    .catch(frase => console.log(frase + '!!!\nHouve o erro descrito acima, queira se atentar para resolvê-lo!')) //catch() será executado só se o promisse der errado...

console.log('Esse execução é aleatória e fora do promise...') //Esse console.log() é independente do promise, sendo assim, ele será executado antes, de forma síncrona, enquando o promise será executado depois, respeitando o seu aspecto assícrono...




//EXEMPLO FICTÍCIO DA VANTAGEM DE SE USAR PROMISES:

//DESVANTANGEM DE UMA CALLBACK HELL SOBRE UM PROMISE:

//Para mostrarmos como um promise é muito mais vantajoso do que utilizar callbacks uma encadeada sobre a outra somente para trazer um resultado, é muito mais vantajosa.
//Abaixo temos 3 URLS que trazem detalhes sobre 3 turmas de uma escola, as turmas A, B e C:

//Endereço do arquivo JSON para Turma A: http://files.cod3r.com.br/curso-js/turmaA.json
//Endereço do arquivo JSON para Turma A: http://files.cod3r.com.br/curso-js/turmaB.json
//Endereço do arquivo JSON para Turma A: http://files.cod3r.com.br/curso-js/turmaC.json

//Queremos gerar uma função que retorne somente os nomes dos alunos de cada turma em um único array...
```

```
//EXEMPLO COM CALLBACKS (Callback Hell>:C):
//Primeiro temos que importar uma biblioteca para conexão com o arquivo por através de protocolo "http"...
const http = require('http') //Estamos usando o módulo interno "http" do node invez de usar o "axios" por que o axios já é baseado em promise, então não faria sentido fazer um exercício com promise numa biblioteca que já utiliza "promise", por isso estamos usando a biblioteca "http" que é mais genérica.

//Abaixo temos a função que captura o arquivo JSON de acordo com a letra da turma...
const getTurma = (letra, callback) => { //Já que o único parâmetro que diferencia as URLS das turmas é uma letra, nossa função vai receber uma letra que será fundida a url de busca por através de um template string. E como 2º parâmetro, vai receber uma callback para o tratamento dos valores posteriormente...
    const url = `http://files.cod3r.com.br/curso-js/turma${letra}.json` //Aqui juntamos a letra com a URL...
    http.get(url, res => { //Usando o método get da biblioteca "http" buscamos uma resolução para a nossa url e também executamos uma função callback que recebe como parâmetro os dados que estiverem no endereço da URL em formato de memória em buffer...
        let resultado = '' //Temos uma variável que vai receber os dados...

        res.on('data', dados => { //a variável vai receber os dados JSON dentro de um array com os dados da URL graças ao "on('data')""
            resultado += dados
        })

        res.on('end', () => { //Aqui temos um evento que finaliza a execução por chamar a callback que irá tratar os dados recebidos, mas antes irá transformar esses dados para o formato JSON...
            callback(JSON.parse(resultado))
        })
    })
}

let nomes = [] //Aqui temos a variável que vai receber somente os nomes de todos os alunos de todas as turmas...
getTurma('A', alunos => { //Perceba que chamamos a função "getTurma" primeiramente passando como parâmetro a letra "A", para pegar os alunos somente da turma "A", e usamos uma callback que mapeia somente os nomes e os atribuí a variável nomes...
    nomes = nomes.concat(alunos.map(a => `A: ${a.nome}`))
    getTurma('B', alunos => { //Em seguida temos outra chamada para a função "getTurma" dentro da chamada da para a turma "A", mas agora chamando a turma "B", só para não ter que chamar a turma B numa segunda chamada separada...
        nomes = nomes.concat(alunos.map(a => `B: ${a.nome}`))
        getTurma('C', alunos => { //E outra para a turma "C"
            nomes = nomes.concat(alunos.map(a => `C: ${a.nome}`))
```

```
            console.log(nomes) //Como podemos ver, todas as chamadas são atribuídas para a mesma variável "nomes"...
        })
    })
})



//REFATORANDO A FUNÇÃO ACIMA USANDO PROMISE USANDO PROMISE:
const getTurmaPromise = letra => { //Perceba que agora, a função recebe somente 1 parâmetro que é a letra correspondente a turma...
    const url = `http://files.cod3r.com.br/curso-js/turma${letra}.json` //Aqui juntamos a letra com a URL...
    return new Promise((resolve, reject) => { //Perceba que agora estamos gerando um promise no retorno da função...
        http.get(url, res => {
            let resultado = '' //Temos uma variável que vai receber os dados...

            res.on('data', dados => { //a variável vai receber os dados JSON dentro de um array com os dados da URL graças ao
"on('data')""
                resultado += dados
            })

            res.on('end', () => { //o evento "end" só ativado quando o evento anterior termina, depois que resultado colhe todos os
dados, ele é ativado
                try { //Mas nessa vez estamos usando um try e catch no evento "end", onde, caso o evento dê certo ele transforma os
dados em JSON, caso dê errado ele retorna a mensagem de erro por através de catch...
                    resolve(JSON.parse(resultado))
                } catch(e){
                    reject(e)
                }
            })
        })
    })

}
```

```
Promise.all([getTurmaPromise('A'), getTurmaPromise('B'), getTurmaPromise('C')]) //Usando o objeto "promise" temos o método "all()", esse método recebe um array de funções para serem executadas, ele vai executar todas as funções e só quando terminar as execuções ele vai dar seguimento ao código...
    .then(turmas => [].concat(...turmas)) //Perceba que o then() pega todos os resultados das execuções e os concatena num array...
    .then(alunos => alunos.map(alunos => alunos.nome)) //O array com os resultados é mapeado e são recolhidos só os nomes em um novo array...
    .then(nomes => console.log('\nRESULTADO COM PROMISSE: VEJA QUE É IGUAL AO RESULTADO DA CALLBACK HELL, E É ATÉ EXECUTADA ANTES POR SER MAIS LEVE...', nomes)) //Finalmente o array somente com os nomes é devolvido...

//Veja o resultado se der erro...
getTurmaPromise('D').catch(e => console.log('\nERRO!!!: ', e.message)) //Usamos a letra para um turma que não existe...




//SE UM ERRO ACONTECER NO THEN() O CATCH() É ACIONADO:
//Veja que temos uma função que gera um número randômico de 0 a 0.9, caso o número passado como parâmetro para a função seja menor que 0.9 a função irá executar um error...
function numbersRandom(value, number) {
    return new Promise((resolve, reject) => {
        if(number < Math.random()){
            reject('Ocorreu um erro')
        } else {
            resolve(value)
        }
    })
}

numbersRandom('Testando...', 1) //Passamos como parâmetro o número 1, isso significa que a função nunca executará um erro...
    .then(v => consol.log(v)) //Perceba que o erro acontece dentro do then(), chamamos um "consol" em vez de "console"
    .catch(err => console.log('\nO erro foi', err)) //Veja que o erro é acionado...
    .then( _ => console.log('Eu ainda aconteço por que eu sou um then()...')) //Veja também a importância de sempre colocar o cacth() no final de uma execução de promise, afinal, qualquer execução then que existir depois de um "catch" será executada...
```

RESULTADO NO CONSOLE...

```
[Running] node "c:\Users\Almoxarifado\Documents\javascript\arquivos_das_aulas\155-Promises.js"
Esse execução é aleatória e fora do promise...

O erro foi ReferenceError: consol is not defined
    at c:\Users\Almoxarifado\Documents\javascript\arquivos_das_aulas\155-Promises.js:135:23
    at processTicksAndRejections (internal/process/task_queues.js:95:5)
Eu ainda aconteço por que eu sou um then()...

ERRO!!!:  Unexpected token < in JSON at position 0

RESULTADO COM PROMISSE: VEJA QUE É IGUAL AO RESULTADO DA CALLBACK HELL, E É ATÉ EXECUTADA ANTES POR SER MAIS LEVE... [
  'Kellia',   'Hi',     'Inge',
  'Myrle',    'Doreen', 'Pennie',
  'Faye',     'Leena',  'Taylor',
  'Juieta',   'Rossie', 'Mary',
  'Dionysus', 'Myca',   'Sharlene',
  'Meghan',   'Perice', 'Micheil',
  'Nat',      'Bone',   'Kellina',
  'Barrie',   'Darda',  'Rainer',
  'Joan',     'Kasper', 'Sammie',
  'Scott',    'Kiel',   'Dell'
]
[
  'A: Kellia',   'A: Hi',     'A: Inge',
  'A: Myrle',    'A: Doreen', 'A: Pennie',
  'A: Faye',     'A: Leena',  'A: Taylor',
  'A: Juieta',   'B: Rossie', 'B: Mary',
  'B: Dionysus', 'B: Myca',   'B: Sharlene',
  'B: Meghan',   'B: Perice', 'B: Micheil',
  'B: Nat',      'B: Bone',   'C: Kellina',
  'C: Barrie',   'C: Darda',  'C: Rainer',
  'C: Joan',     'C: Kasper', 'C: Sammie',
  'C: Scott',    'C: Kiel',   'C: Dell'
]
Essa frase veio de um promise... Sim ela veio, mas só depois de 2 segundos
```

```
[Done] exited with code=0 in 2.117 seconds
```