

Dell IT Academy

HTML, CSS e TypeScript

Instrutor: Júlio Pereira Machado (julio.machado@pucrs.br)



Módulos



Módulos

- É extremamente conveniente dividir e organizar código em módulos
- Um módulo é um agrupamento de código que provê funcionalidade para outros módulos utilizarem (sua interface) e especifica outros módulos que ele utiliza (suas dependências)
- Benefícios:
 - Facilita a organização e a distribuição de blocos de funções e objetos relacionados
 - Permite a reutilização de código
 - Provê um “espaço de nomes” para evitar o compartilhamento de variáveis globais
- Diferentes padrões para a implementação de módulos:
 - *CommonJS*
 - *Asynchronous Module Definition*
 - *Universal Module Definition*
 - *ECMAScript Modules*
 - *etc*

Módulos - CommonJS

- Padrão utilizado por um grande número de pacotes disponibilizados via NPM
- Ambiente de execução do NodeJS suporta o padrão CommonJS
- Módulos definem suas interfaces via *exports* e *module.exports*
 - Use *exports* para adicionar propriedades ao objeto criado automaticamente pelo sistema de módulos
 - Use *module.exports* para definir o próprio objeto a ser retornado
- Dependências para outros módulos são importadas via *require*

Módulos - CommonJS

- Definição do módulo: exportando funções no objeto padrão

```
exports.area = (r) => Math.PI * r**2;  
exports.circunferencia = (r) => 2 * Math.PI * r;
```

- Importando o módulo:

```
const circulo = require('./circulo_funcoes');  
console.log(`Área do círculo de raio 4 é ${circulo.area(4)}`);  
  
//desestruturando o objeto e acessando a função diretamente  
const {area} = require('./circulo_funcoes');  
console.log(`Área do círculo de raio 2 é ${area(2)}`);
```

Módulos - CommonJS

- Definição do módulo: exportando objeto

```
module.exports = class Circulo {  
  constructor(r) {  
    this.raio = r;  
  }  
  area() {  
    return Math.PI * this.raio**2;  
  }  
  circunferencia() {  
    return 2 * Math.PI * this.raio;  
  }  
};
```

Módulos - CommonJS

- Importando o módulo:

```
const Circulo = require('./circulo_objeto');  
const c1 = new Circulo(4);  
console.log(`Área do círculo de raio 4 é ${c1.area()}`);
```

Módulos - CommonJS

- Definição do módulo: exportando objeto (notação TypeScript)

```
class Circulo {  
  constructor(r) {  
    this.raio = r;  
  }  
  area() {  
    return Math.PI * this.raio**2;  
  }  
  circunferencia() {  
    return 2 * Math.PI * this.raio;  
  }  
};  
  
exports = Circulo;
```


Módulos - CommonJS

- Importando o módulo:

```
import Circulo = require('./circulo_objeto');  
const c1 = new Circulo(4);  
console.log(`Área do círculo de raio 4 é ${c1.area()}`);
```

Módulos – ES

- Padrão nativo do JavaScript disponível a partir do ECMAScript 6 (2015)
 - TypeScript suporta módulos ES6
 - Qualquer arquivo contendo *import* ou *export* (de nível mais alto) é considerado um módulo

Módulos – ES

- Módulos definem suas interfaces via palavra-chave *export*
 - Qualquer declaração pode ser exportada adicionando-se *export*
 - Vinculação de exportação *default* é tratado como elemento principal do módulo
 - Comandos *export {}* podem ser utilizados para renomear os elementos exportados
 - Comandos *export {} from* podem ser utilizados para reexportar elementos

Módulos – ES

- Dependências para outros módulos são importadas via palavra-chave *import*
 - Importar um nome a partir do módulo, importa a exportação *default*
 - Importar com sintaxe de desestruturação {} permite importar elementos indicados
 - Importar com * importa o módulo inteiro
 - Importações com {} ou * permite modificar o nome do que foi importado via operador *as*

Módulos – ES

- Definição do módulo (circulo_funcoes.ts): exportando funções no objeto padrão

```
export function area(r: number): number { return Math.PI * r ** 2; }  
export function circunferencia(r: number): number { return 2 * Math.PI * r; }
```

- Importando o módulo (index.ts):

```
import { area, circunferencia as circ } from "./circulo_funcoes";  
console.log(`Área do círculo de raio 4 é ${area(4)}`);  
console.log(`Circunferência do círculo de raio 4 é ${circ(4)}`);  
  
import * as circulo from "./circulo_funcoes";  
console.log(`Área do círculo de raio 2 é ${circulo.area(2)}`);  
console.log(`Circunferência do círculo de raio 4 é ${circulo.circunferencia(4)}`);
```

Módulos – ES

- Definição do módulo (circulo_objeto.ts): exportando objeto

```
export default class Circulo {  
  constructor(public raio: number){  
  }  
  area(): number {  
    return Math.PI * this.raio ** 2;  
  }  
  circunferencia(): number {  
    return 2 * Math.PI * this.raio;  
  }  
}
```

Módulos – ES

- Definição do módulo (circulo_objeto.ts): exportando objeto

```
class Circulo {  
  constructor(public raio: number){  
  }  
  area(): number {  
    return Math.PI * this.raio ** 2;  
  }  
  circunferencia(): number {  
    return 2 * Math.PI * this.raio;  
  }  
}  
  
export {Circulo};
```

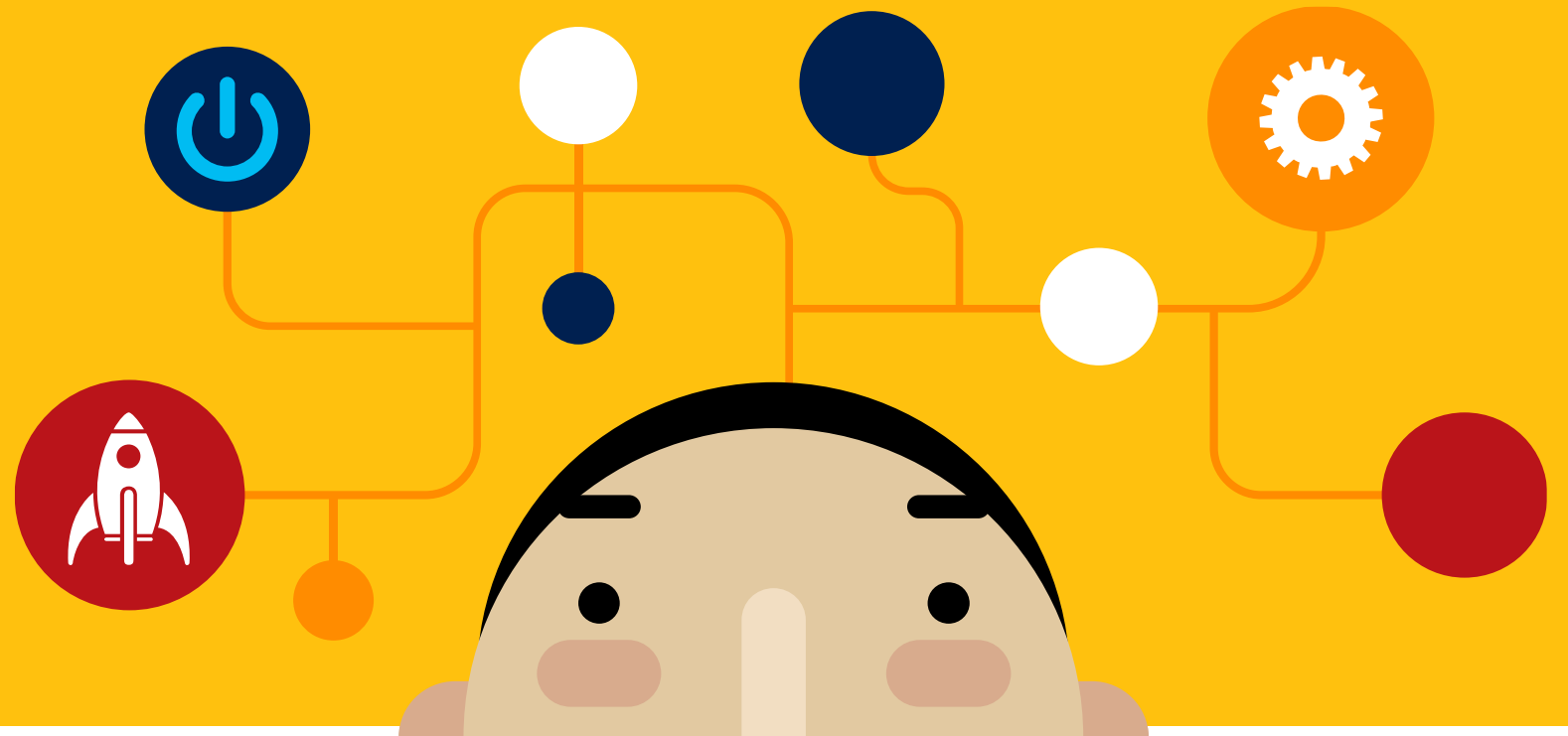
Módulos – ES

- Importando o módulo (index.ts):

```
import Circulo from "./circulo_objeto";  
let circ: Circulo = new Circulo(4);  
console.log(`Área do círculo de raio 2 é ${circ.area()}`);  
console.log(`Circunferência do círculo de raio 4 é ${circ.circunferencia()}`);
```


Laboratório

- Abra as instruções do arquivo Lab04_TypeScript_Outros



Exceções



Exceções

- Falhas nas condições podem ser indicadas ao programador através do conceito de exceções
- Quando uma função encontra uma situação anormal, ele informa tal anormalidade pelo lançamento (geração) de uma exceção
 - Ex.: a função `JSON.parse(string)`, irá lançar uma exceção `SyntaxError` se o formato do objeto JSON for incorreto
- Quando um bloco de código tenta detectar uma situação anormal, ele captura essa exceção, possivelmente indicando que irá realizar o tratamento do problema encontrado

Lançando Exceções

- Para lançar uma exceção dentro de uma função que estamos desenvolvendo:
 - Lançar a exceção via comando *throw*
 - Utilizar objetos *Error* e suas subclasses
 - Propriedades principais: *name*, *message* e *stack*

Novas Exceções

- Para criar novos tipos de exceções, podemos criar subclasses de *Error*
- Exemplo:

```
class ValidationError extends Error {  
  constructor(message) {  
    super(message); // construtor da superclasse  
    this.name = "ValidationError"; // alterando propriedade padrão de Error  
  }  
}  
  
function vaiDarErro() {  
  throw new ValidationError("Dados inválidos!");  
}
```

Capturando Exceções

- Para capturar e tratar exceções, utiliza-se o bloco de comandos *try...catch...finally*
 - No bloco *try* estão colocados os comandos que podem provocar o lançamento de uma exceção
 - As exceções são capturadas no bloco *catch*
 - O bloco *finally* contém código a ser executado, independente da ocorrência de exceções

```
try
{
    // código que pode gerar exceção
}
catch (e)
{
    // código que trata exceção
}
finally
{
    // tratamento geral
}
```

Capturando Exceções

- Bloco *catch* captura todas exceções
 - Uma técnica é tratar as exceções adequadas ao momento e relançar as demais que não se sabe como tratar no momento

```
let json = {incorreto};
try {
  let pessoa = JSON.parse(json);
  console.log(pessoa.nome);
} catch(err) {
  if (err instanceof SyntaxError) {
    //tratar a exceção
  } else {
    throw err; //relançar a exceção não-tratada
  }
}
```

Funções Assíncronas



Programação assíncrona em JavaScript

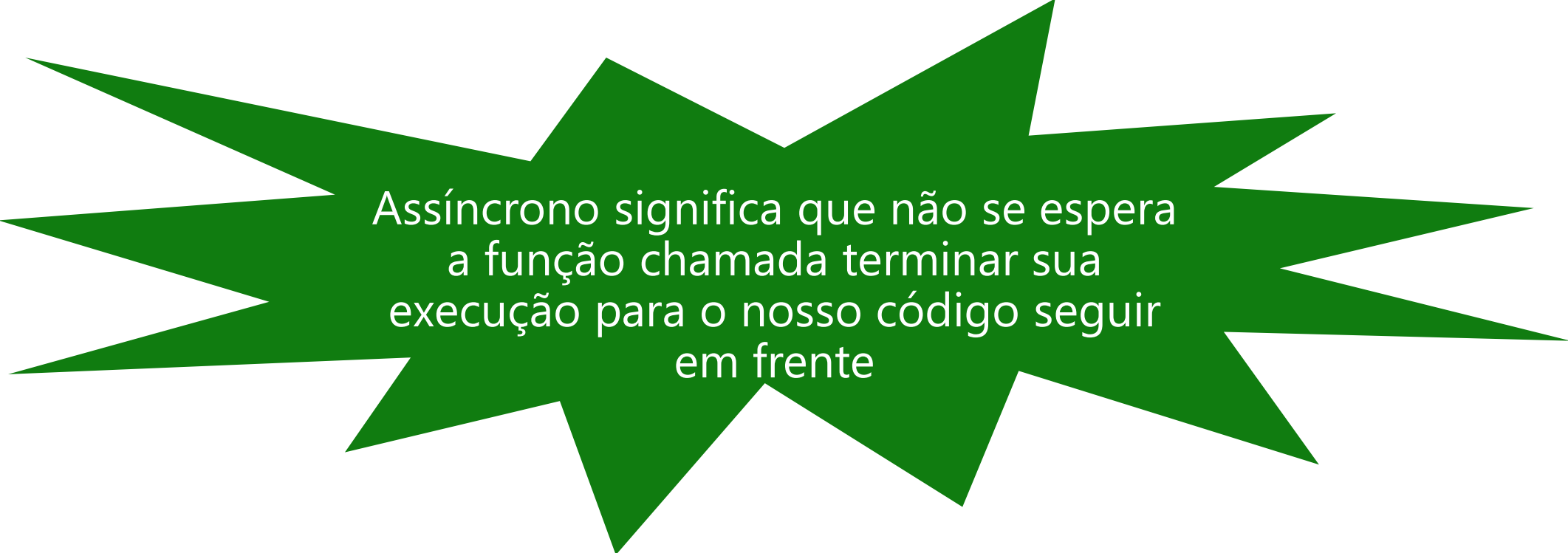
Callbacks

Promises

Async/Await

Programação assíncrona em JavaScript

- A API de programação do JavaScript possui muitas funções de execução assíncrona
 - Por exemplo, o pacote "fs" do NodeJS possui muitas funções para manipulação de arquivos de maneira assíncrona



Assíncrono significa que não se espera a função chamada terminar sua execução para o nosso código seguir em frente

Callbacks

Callback é o nome que se dá a uma função passada como argumento para uma segunda função de alta-ordem, na qual essa segunda função irá executar a primeira função em algum momento futuro.

- Se esse tempo futuro for antes do término da segunda função, então o *callback* é considerado síncrono.
- Se esse tempo futuro for depois do término da segunda função, então o *callback* é considerado assíncrono.

Callbacks

Muitas APIs de JavaScript para funções assíncronas utiliza o conceito de funções de callback

- Exemplos: *setTimeout()*, *setInterval()*, módulo *node:fs*

Resulta em pequenas funções que são encadeadas para realizar um processamento

- CUIDADO! Encadear múltiplos callbacks resulta em um código de difícil manutenção.

Callbacks - Exemplo

```
import * as fs from 'fs';  
  
fs.readFile('package.json', function (err, buf) {  
    console.log(buf.toString());  
});
```

`fs.readFile(path[, options], callback)`



Callbacks - Exemplo

```
import * as fs from 'fs';

fs.readFile('package.json', function (err, buf) {
  if (err) {
    //tratar o erro aqui
  } else {
    console.log(buf.toString());
  }
});
```

Callbacks - Exemplo

```
import * as fs from 'fs';  
  
fs.readFile('package.json', function (err, buf) {  
    if (err) throw err; //não sei como tratar aqui  
    console.log(buf.toString());  
});
```

Callbacks - Exemplo

```
import * as fs from 'fs';


const onRead = function (err, buf) {
  console.log(buf.toString());
};

fs.readFile('package.json', onRead);
```


Callbacks - Encadeamento

```
meuCofre.salvar(  
  function (err, dados) {  
    console.log('Dados armazenados: ' + dados);  
  }  
);
```

```
minhasContas.buscarTotal(  
  function (err, dados) {  
    console.log('Total: ' + dados);  
  }  
);
```



Como fazer para a
função buscarTotal()
somente ser executada
depois de salvar()?

Callbacks - Encadeamento

```
meuCofre.salvar(  
  function (err, dados) {  
    console.log('Dados armazenados: ' + dados);  
    minhasContas.buscarTotal(  
      function (err, dadosTotal) {  
        console.log('Total: ' + dadosTotal);  
      }  
    );  
  }  
);
```

Callbacks - Encadeamento

```
meuCofre.salvar(  
  function (err, dados) {  
    onSalvar(err, dados);  
  }  
);  
  
const onSalvar = function (err, dados) {  
  console.log('Dados armazenados: ' + dados);  
  minhasContas.buscarTotal(  
    function (err, dadosTotal) {  
      onBuscarTotal(err, dadosTotal);  
    }  
  );  
};  
  
const onBuscarTotal = function (err, dadosTotal) {  
  console.log('Total: ' + dadosTotal);  
};
```

Callbacks - Encadeamento

```
var fs = require('fs');

fs.readdir('.', function (err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function (filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function (err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function (width, widthIndex) {
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(destination + 'w' + width + '_' + filename, function (err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }).bind(this))
        }
      })
    })
  }
})
```



Desvantagens de *callbacks*

- *Callbacks* não possuem um padrão para a indicação de ocorrência de erros na computação assíncrona.
 - Cada API tem uma forma própria de lidar com a questão.
 - Usualmente utilizam a convenção do primeiro parâmetro da função de *callback* ser o objeto que contém a informação do erro ocorrido.

Promises

- A partir do ECMAScript6 (2015), a linguagem fornece o suporte a objetos *Promise*
- Permitem o controle do fluxo de execução assíncrono de funções de maneira mais “limpa” do que o uso de *callbacks*
- Representa o resultado final ou falha de uma operação assíncrona
- Ideia: uma função irá retornar uma promessa de um objeto contendo o resultado de interesse no futuro

Promises

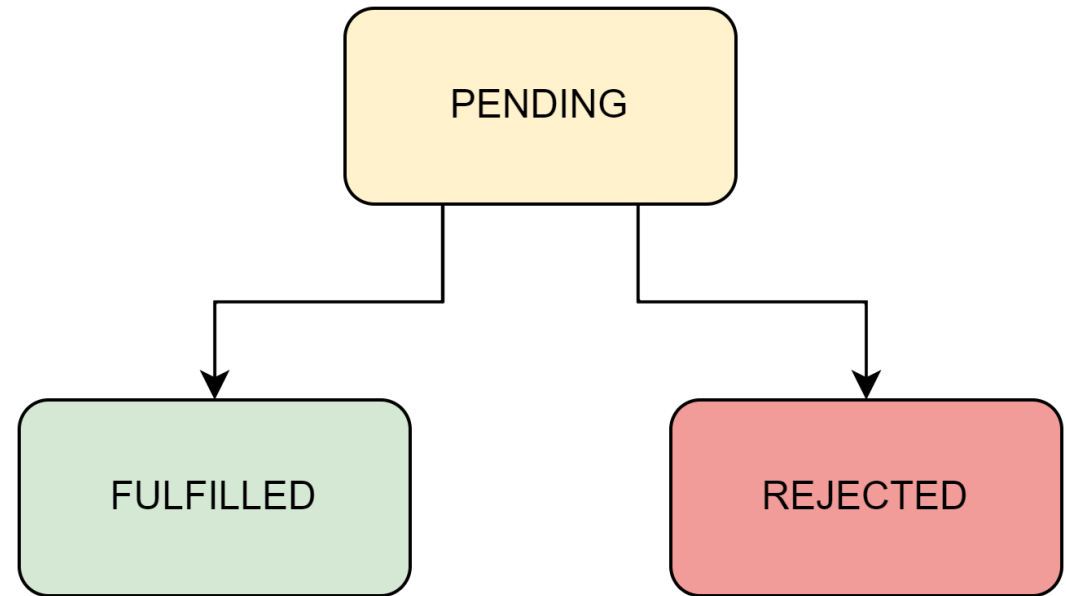
Promise é um objeto *proxy* que encapsula um código com comportamento assíncrono e atua como um repositório para um resultado que ainda não existe.

IMPORTANTE:

- *Promises* não tornam um código assíncrono por si só.
- Elas provêm um meio de observar o resultado de um código que já é assíncrono.

Estado de uma *promise*

- Uma *promise* possui três estados possíveis:
 - *Pending* – é o estado inicial, indicando que a execução assíncrona ainda não iniciou ou ainda está em progresso;
 - *Fulfilled* – quando a *promise* foi “decidida” com um possível valor como resultado, usualmente indicando uma computação que completou com sucesso;
 - *Rejected* – quando a *promise* foi “decidida” com um possível erro como razão de rejeição, usualmente indicando uma computação que completou com falha.



Estado de uma *promise*

- O estado interno de uma *promise* está encapsulado e não pode ser externamente inspecionado nem alterado.
- Uma vez que a *promise* seja "decidida" como *fulfilled* ou *rejected*, ela não muda mais de estado.
 - Ou seja, as únicas transições de estado possíveis são:
 - *pending* → *fulfilled*
 - *pending* → *rejected*
- CUIDADO! Não existe nenhuma garantia que uma *promise* irá sair do estado *pending* no futuro!

Criação de *promises*

- Construtor de objetos Promise:

```
new Promise( (resolve, reject) => {  
  // corpo do executor com comportamento assíncrono  
});
```

- Função passada ao construtor é conhecida como “executor” e é acionada automaticamente de forma síncrona quando a *promise* é criada.
- Essa função define o comportamento assíncrono que será executado e que “no futuro” irá retornar um valor ou um erro.

Criação de *promises*

- Observação:
 - Criar *promises* diretamente é uma tarefa rara para o dia a dia do programador JavaScript.
 - Grande parte do tempo estamos consumindo APIs assíncronas que já nos fornecem as *promises*.
 - Contudo, ao criarmos nossas próprias APIs assíncronas, a necessidade de criação das *promises* pode surgir e devemos saber como agir.

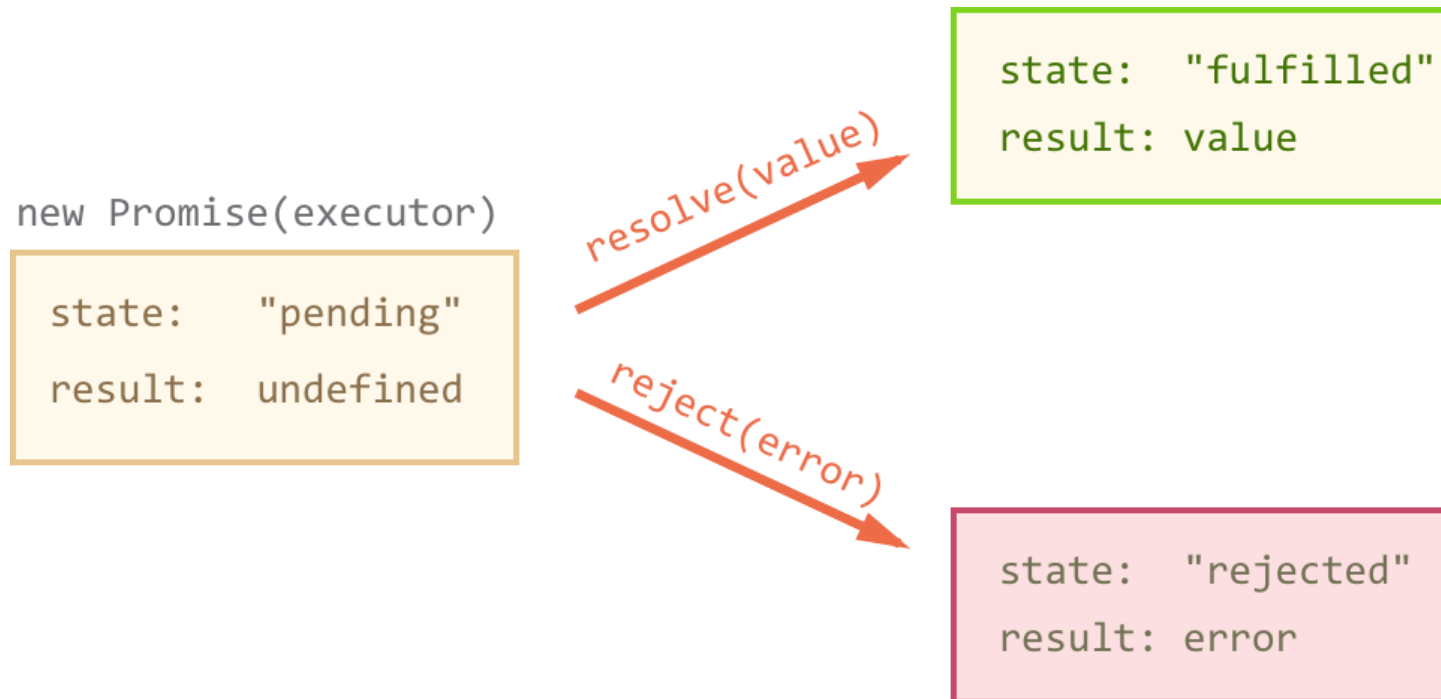
Manipulação de *promises*

- O estado da *promise* somente pode ser manipulado internamente através das duas funções que são parâmetros da função executora:
 - *resolve* – “decide” a *promise* para o estado *fulfilled*.
 - Opcionalmente recebe um argumento de entrada que corresponde ao valor do resultado da computação assíncrona.
 - *reject* – “decide” a *promise* para o estado *rejected*.
 - Opcionalmente recebe um argumento de entrada que corresponde à razão da falha da computação assíncrona.

```
new Promise( (resolve, reject) => {  
  // corpo do executor com comportamento assíncrono  
});
```

Manipulação de *promises*

- A ação de um objeto promise pode:
 - Terminar com sucesso – diz-se que a promise foi “resolvida” e está no estado “fulfilled”
 - Executar a função *resolve(valor)*
 - Terminar com falha – diz-se que a promise foi “rejeitada” e está no estado “rejected”
 - Executar a função *reject(erro)*





Exemplo

- Observando o estado de uma *promise*.

```
const p = new Promise(() => {});  
setTimeout(console.log, 0, p);
```

```
Promise { <pending> }
```



Exemplo

- Observando o estado de uma *promise*.

```
const p = new Promise((resolve, reject) => {  
    setTimeout(resolve, 3000, "resultado");  
});  
setTimeout(console.log, 0, p);  
setTimeout(console.log, 4000, p);
```

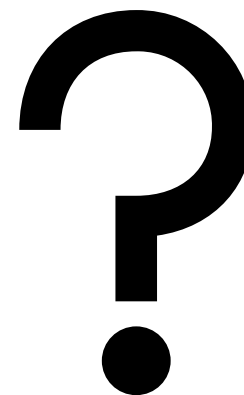
```
Promise { <pending> }  
Promise { 'resultado' }
```



Exemplo

- Observando o estado de uma *promise*.

```
const p = new Promise((resolve, reject) => {  
    setTimeout(reject, 1000, new Error("falha"));  
});  
setTimeout(console.log, 0, p);  
setTimeout(console.log, 3000, p);
```



Interagindo com uma *promise*

- Um objeto *promise* possui três métodos para o registro de funções tratadoras (ou *callbacks*):
 - *then()* – adiciona um tratador a ser acionado quando a *promise* transita para o estado *fulfilled*;
 - *catch()* – adicionar um tratador a ser acionado quando a *promise* transita para o estado *rejected*;
 - *finally()* – adiciona um tratador a ser acionado quando a *promise* é “decidida”, não importa se com sucesso ou falha.
- IMPORTANTE: os três métodos retornam uma nova *promise* cujo estado é derivado do estado da *promise* original, permitindo uma técnica de “encadeamento” de *promises*.

Interagindo com uma *promise*: *then*

- Esse método pode registrar duas funções de tratamento invocadas de maneira assíncrona:
 - Uma para tratar o resultado da computação com sucesso;
 - Uma para tratar o resultado da computação com falha.
- São mutuamente exclusivos.

```
promise.then(  
  function(result) { /* tratar o resultado com sucesso  
*/ },  
  function(error) { /* tratar o resultado com erro */ }  
);
```

```
let promise = new Promise(...);  
promise.then(  
  result => console.log(result),  
  error => console.log(error)  
);  
promise.then(  
  result => console.log(result)  
);  
promise.then(  
  undefined,  
  result => console.log(result)  
);
```

Interagindo com uma *promise*: *catch*

- Esse método registra uma função de *callback* que será invocada de maneira assíncrona quando o objeto *promise* for rejeitado.
- É apenas uma forma alternativa para o método *then(undefined, callback)*.

```
let promise = new Promise(...);  
  
promise.catch(  
  error => console.log(error)  
);
```

Interagindo com uma *promise*: *finally*

- Esse método registra uma função de *callback* que será invocada de maneira assíncrona quando o objeto *promise* for “decidida”, tanto com sucesso quanto com falha.
- Útil para realizar código que deveria ser executado em ambos os casos, evitando duplicação.

```
let promise = new Promise(...);  
  
promise.finally(  
    () => console.log("finalização")  
);
```

Encadeamento de *promises*

- Utilizar os métodos *then()*, *catch()* e *finally()* permite a sequencialização de chamadas de funções assíncronas.
- O padrão de codificação é que o tratamento do *callback* registrado produz um resultado que é uma outra *promise* passada adiante.

```
let promise = new Promise( (resolve, reject) => { ... } );

promise
  .then(
    result => { console.log(result); return 'valor'; })
  .then(
    result => console.log(result)
  )
  .catch(
    error => console.log(error)
  );
```

Promises

- Para obter o resultado de uma promise, utiliza-se o método *then*
 - Esse método registra uma função de *callback* que será chamada quando o objeto promise produz um resultado

```
promise.then(  
  function(result) { /* tratar o resultado com sucesso */ },  
  function(error) { /* tratar o resultado com erro */ }  
);
```

```
let promise = new Promise(function(resolve, reject) { ... });
```

```
promise.then(  
  result => console.log(result),  
  error => console.log(error)  
);
```

```
promise.then(  
  result => console.log(result)  
);
```

Promises

- Para tratar de uma promise rejeitada utiliza-se o método *catch*
 - Esse método registra uma função de *callback* que será chamada quando o objeto promise produz algum tipo de exceção
 - É apenas um alias para o método *then(null, callback)*

```
let promise = new Promise(function(resolve, reject) { ... });

promise.catch(
  error => console.log(error)
);
```

Async/Await

- Disponível a partir do ECMAScript 2017.
- Modelo para facilitar a escrita de código com um “estilo síncrono” mesmo sendo baseado em objetos *Promise*.
- Duas novas palavras-chave:
 - *async* – para a declaração de funções assíncronas que retornam uma *Promise*;
 - *await* – operador utilizado sobre uma *Promise* para obter o resultado de uma computação com sucesso.

Async

- Palavra-chave *async* marca uma função/método como sendo assíncrono.

```
async function funcaoAssincrona() {...}  
const funcaoAssincrona = async function() {...};  
const funcaoAssincrona = async () => {...};  
class Classe {  
  async metodoAssincrono() {...}  
}
```

Async

- Uma função/método assíncrono automaticamente retorna um objeto *Promise* para retornos de qualquer tipo.
- Se uma exceção é gerada dentro de uma função assíncrona, automaticamente será retornado um objeto *Promise* no estado *rejected*.



Exemplo

```
async function funcaoAssincrona() {  
    console.log(1);  
    return 3;  
}  
funcaoAssincrona().then(console.log);  
console.log(2);
```

1

2

3



Exemplo

```
async function funcaoAssincrona() {  
    console.log(1);  
    throw new Error("erro!");  
}  
funcaoAssincrona()  
    .then(console.log)  
    .catch(e => console.error(e.message));  
console.log(2);
```

```
1  
2  
erro!
```

Await

- Palavra-chave *await* antes de uma expressão que fornece um objeto *Promise* faz com que o código espere até que a *promise* seja resolvida (fornecendo um resultado) ou rejeitada (levantando uma exceção).
- *Await* somente pode ser utilizado:
 - No corpo de funções/métodos assíncronos marcados com *async*;
 - No nível mais alto de módulos.

```
async function fazAlgo() {  
  const promise = new Promise( (resolve, reject) => { ... } );  
  const resultado = await promise;  
  return resultado;  
}
```

Await

- Se uma *promise* é rejeitada, o *await* gera uma exceção.

```
async function funcao() {  
  try {  
    let resultado = await umaFuncaoAssincrona();  
    console.log(resultado);  
  } catch(err) {  
    if (err instanceof SyntaxError) {  
      //tratar a exceção  
    } else {  
      throw err; //relançar a exceção não-tratada  
    }  
  }  
}  
  
funcao().catch(erro => console.log(erro));
```



Exemplo

```
async function funcaoAssincrona() {  
    console.log(1);  
    return 3;  
}  
let resultado = await funcaoAssincrona();  
console.log(2);  
console.log(resultado);
```

1

2

3



Exemplo

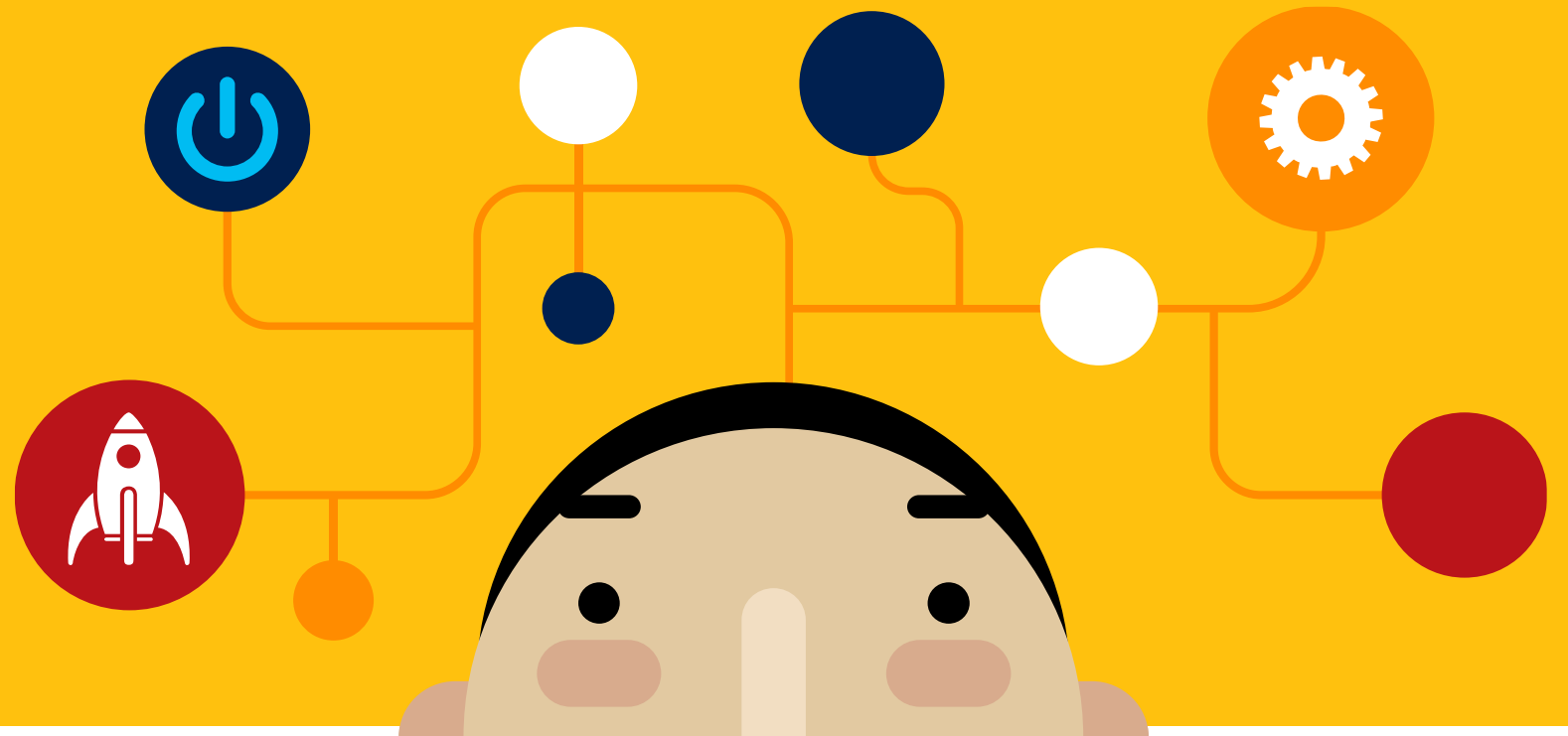
```
async function funcaoAssincrona() {  
    console.log(1);  
    throw new Error("erro!");  
}  
try {  
    let resultado = await funcaoAssincrona();  
    console.log(2);  
    console.log(resultado);  
} catch (e) {  
    console.error(e.message);  
}
```

1

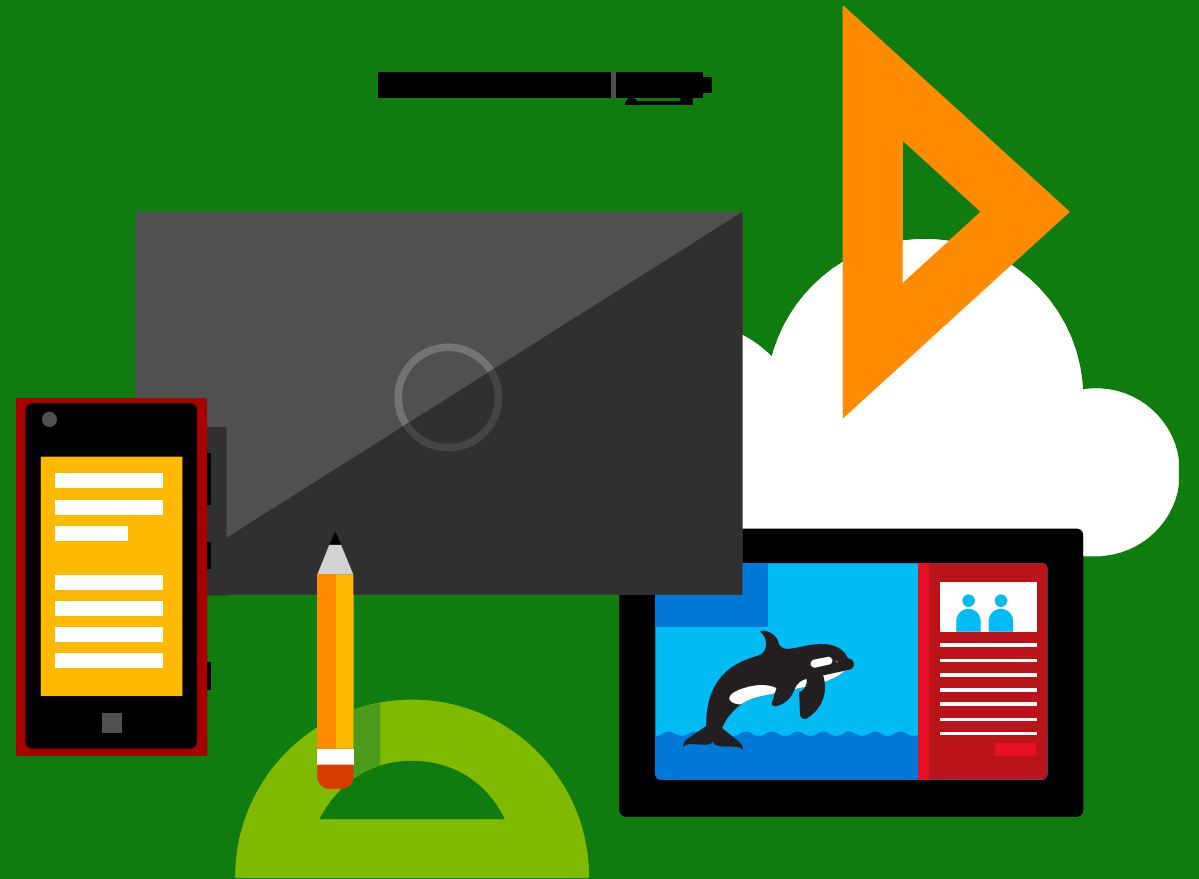
erro!

Laboratório

- Abra as instruções do arquivo Lab04_TypeScript_Outros



Web Services REST



Consumindo Serviços REST

- Um código TypeScript rodando no ambiente Node tem várias opções para consumir serviços REST:
 - Módulo http da API nativa do Node
 - Node-fetch <https://www.npmjs.com/package/node-fetch>
 - Axios <https://axios-http.com/>
 - etc

Consumindo Serviços REST

- Exemplo:
 - Módulo node-fetch <https://www.npmjs.com/package/node-fetch>
 - Instalação:
 - `npm install node-fetch`

Consumindo Serviços REST

- Exemplo:
 - Módulo axios <https://www.npmjs.com/package/axios>
 - Instalação:
 - `npm install axios`

Laboratório

- Abra as instruções do arquivo Lab04_TypeScript_Outros

