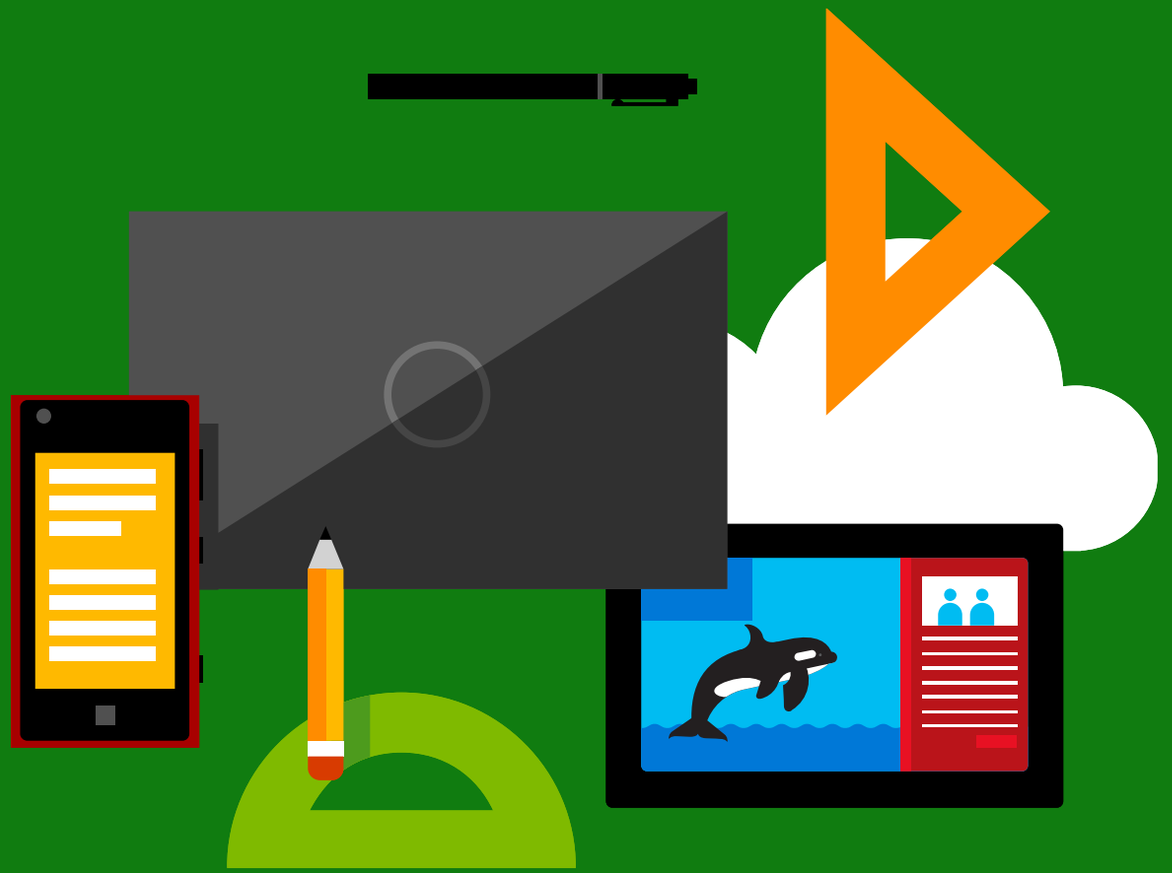
Desenvolvimento Web

HTML, CSS e TypeScript

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

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

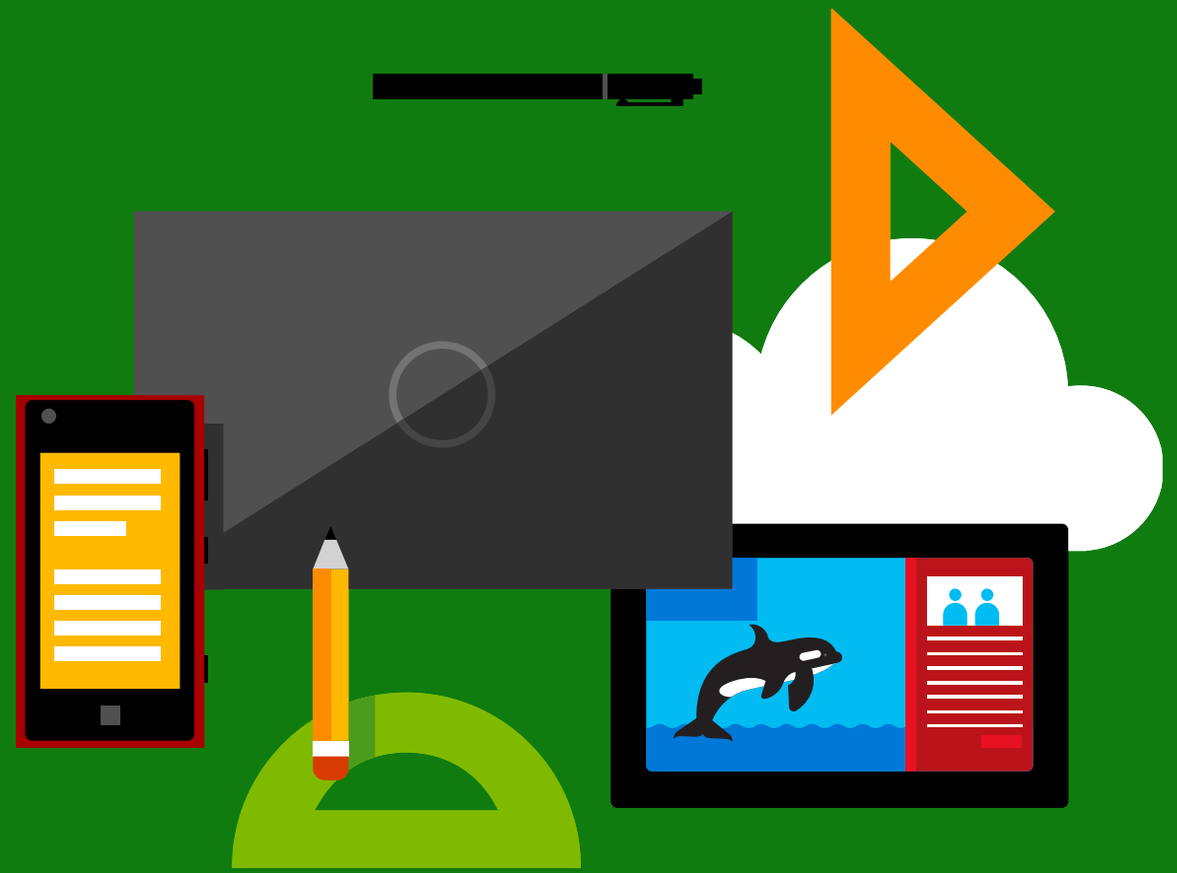
```
try
{
  // código que pode gerar exceção
}
catch (e)
{
  // código que trata exceção
}
finally
{
  // tratamento geral
}
```

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
  }
}
```

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*

Módulo - 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*

Módulo - 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)}`);
```

Módulo - 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;
  }
};
```

Módulo - 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()}`);
```

Módulo - 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;
```

Módulo - 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};
```

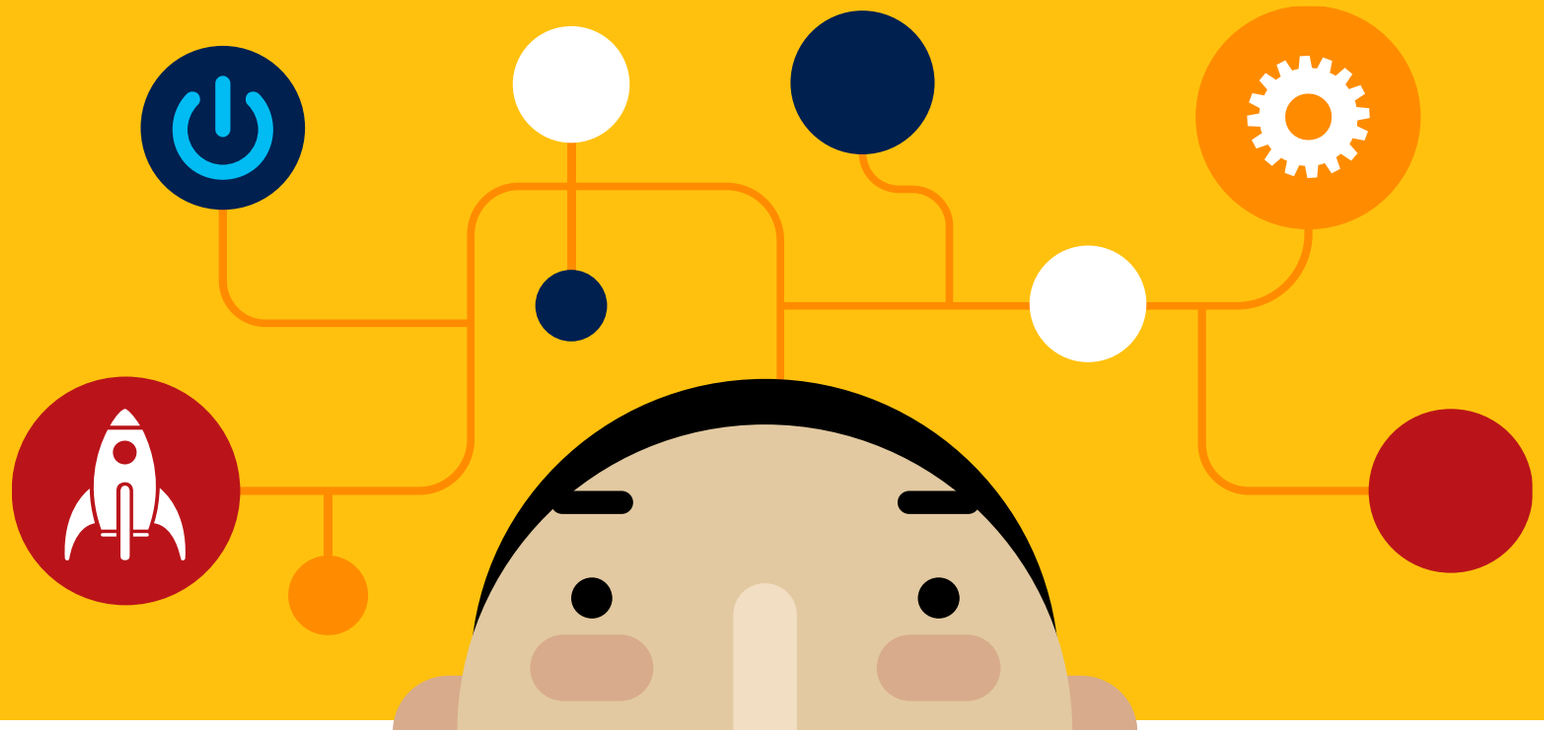
Módulos – ES

- Importando o módulo (index.ts):

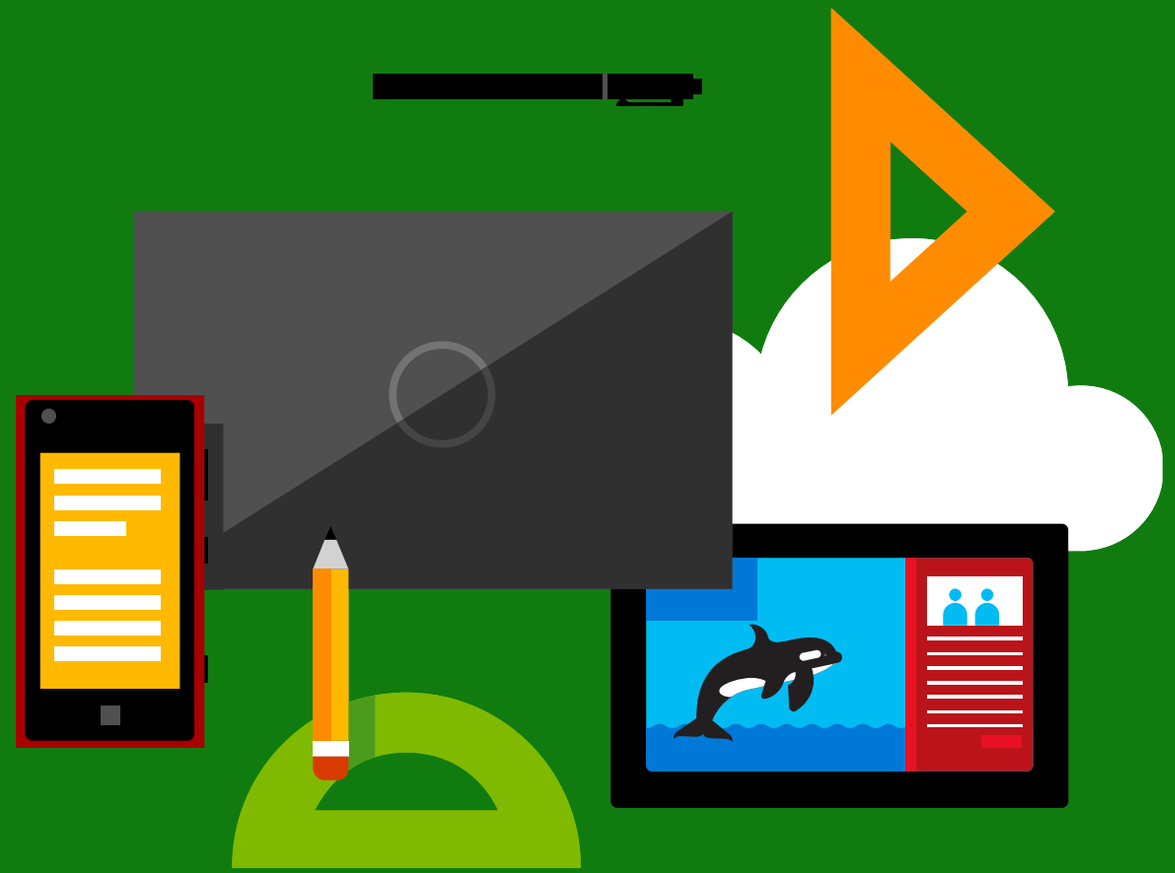
```
import Circulo from "./circulo_objeto";
let circ: Circulo = new Circulo(4);
console.log(`Área do círculo de raio 2 é ${circ.area()}`);
console.log(`Circunferência do círculo de raio 4 é ${circ.circunferencia()}`);
```

Laboratório

- Abra as instruções do arquivo Lab04_TypeScript_Outros

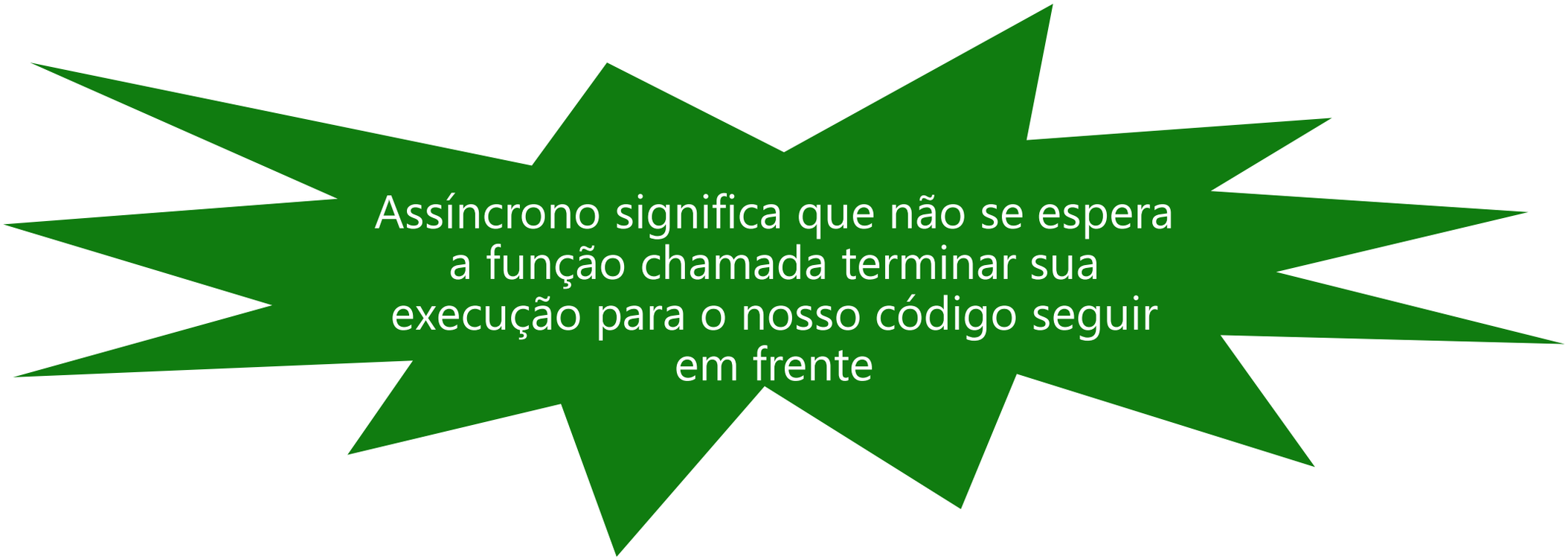


Funções Assíncronas



Funções Assíncronas

- 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



Callbacks

- Muitas APIs de JavaScript para funções assíncronas utiliza o conceito de funções de *callback*
  - São funções que são chamadas quando uma outra função terminou seu processamento
- Resulta em pequenas funções que são encadeadas para realizar um processamento
  - 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 Insano!

```
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))
                }
            })
        })
    }
})
```

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

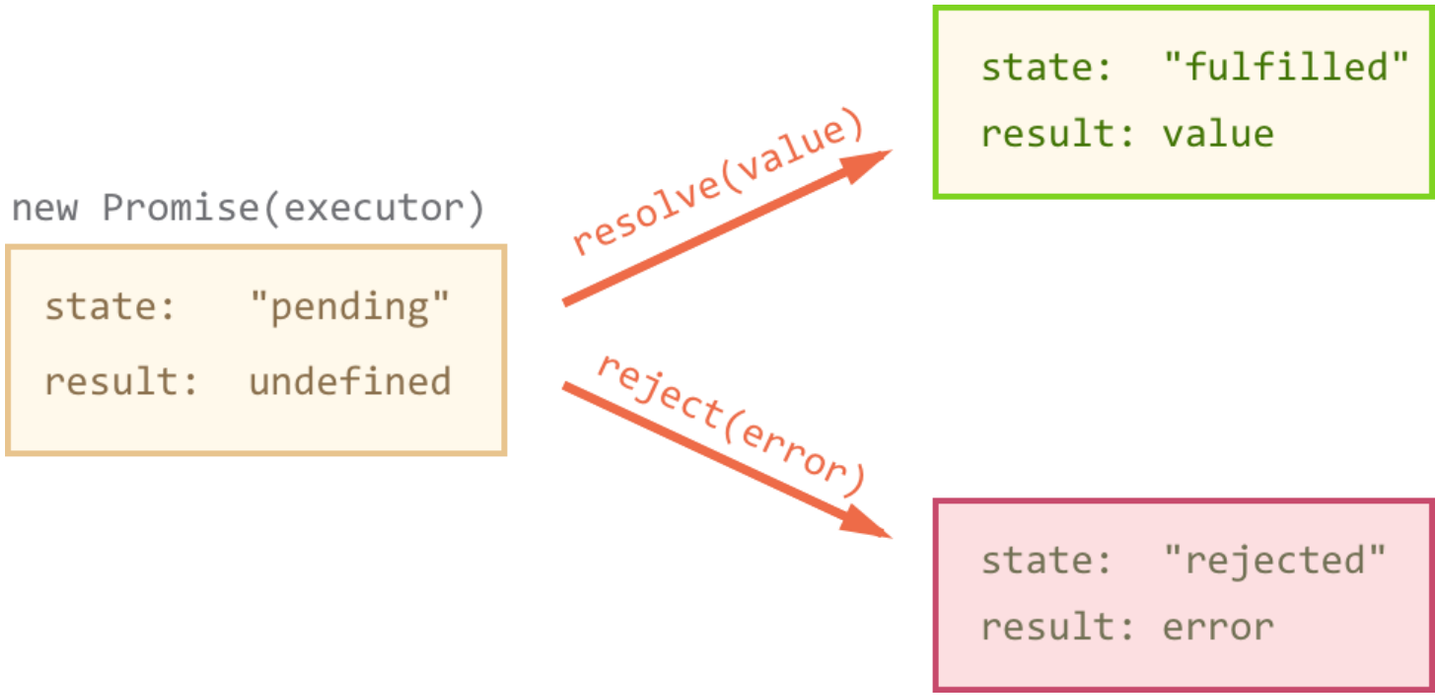
- Construtor de objetos *Promise:*

```
let promise = new Promise(function(resolve, reject) {
  // corpo do executor
});
```

  - Função passada ao construtor é chamada de executor e é chamada automaticamente quando a promise é criada
  - Essa função possui o que será executado e que “no futuro” irá retornar um valor ou um erro
- Propriedades internas da promise (sem acesso público):
  - *state* – representa o estado da execução da promise, inicialmente “pending”
  - *result* – representa o resultado da computação, inicialmente *undefined*

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)*



Promises

- Exemplo:

```
function readFileAsync(filename: string): Promise<any> {
    return new Promise((resolve,reject) => {
        fs.readFile(filename,(err,result) => {
            if (err) reject(err);
            else resolve(result);
        });
    });
}
```

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)
);
```

Promises

- As promises podem ser encadeadas
  - Permite o a sequencialização de chamadas de funções assíncronas
- O padrão de codificação é que o *callback* registrado via método *then* produz um resultado que é uma outra promise passada adiante
  - A função de *callback* pode retornar uma promise configurada por ela mesmo ou um valor qualquer (que é automaticamente encapsulado como o resultado uma promise)

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

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

Promises

- JavaScript permite manipular coleções de promises a fim de implementar diferentes mecanismos de controle de fluxo
  - *Promises.all(iterável)* – retorna uma promise que espera até que todas as promises da coleção iterável tenham terminado com sucesso, retornando um array dos resultados; se uma promise for rejeitada, rejeita toda a coleção
  - *Promises.race(iterável)* – retorna uma promise que espera até que uma das promises da coleção iterável tenha terminado com sucesso, retornando o resultado e ignorando os demais; se uma promise for rejeitada primeiro, rejeita toda a coleção, ignorando as demais

Promises

```
Promise.all([
    asyncFunc1(),
    asyncFunc2(),
])
.then(([result1, result2]) => {
    ...
})
.catch(err => {
    // Recebe primeira rejeição entre as funções
    ...
});
```

Async/Await

- Disponível a partir do ECMAScript 2017
- Modelo sintático para facilitar o uso de objetos *Promise*
- Palavra-chave *async* marca uma função ou método como sendo assíncrono
  - Quando uma função assíncrona for chamada, ela automaticamente retorna um objeto *Promise* para retornos de qualquer tipo
- 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 o resultado) ou rejeitada (levantando uma execção)
  - Só pode ser utilizada dentro de funções marcadas com *async*

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

Async/Await e Exceções

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

```
async function fazAlgo() {
  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
    }
  }
}

fazAlgo().catch(erro => console.log(erro));
```

Async/Await e Exceções

- Versão alternativa utilizando em módulo diretamente

```
try {
    let resultado = await umaFuncaoAssincrona();
    console.log(resultado);
  } catch(err) {
    console.log(err);
  }
}

export{}
```

Laboratório

- Abra as instruções do arquivo Lab04_TypeScript_Outros