



PROGRAMAÇÃO ORIENTADA A OBJETOS

Edson Ifarraguirre Moreno – Aula 03

Professores

ALESSANDRO VALÉRIO DIAS

Professor Convidado

Mestre em Administração e Negócios pela PUCRS, possui também os títulos de especialista em duas áreas: Gerenciamento de Projetos de Tecnologia da Informação e Informática na Educação. É bacharel em Ciência da Computação e Psicologia pela Pontifícia Universidade Católica do Rio Grande do Sul. Atualmente, é analista de sistemas da Pontifícia Universidade Católica do Rio Grande do Sul e professor dos cursos de Análise e Desenvolvimento de Sistemas e de Ciência da Computação do Centro Universitário UniRitter.

EDSON IFARRAGUIRRE MORENO

Professor PUCRS

Doutor em Ciência da Computação pela Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS). Atualmente, é professor adjunto pela mesma PUCRS, estando vinculado a Escola Politécnica, sendo responsável por disciplinas da área de hardware para os cursos de Ciência da Computação, e Engenharia da Computação. Adicionalmente, trabalha com a orientação de alunos no desenvolvimento de projetos do curso de Engenharia de Software. Desde 2016, coordena o laboratório iSeed Labs, uma parceria entre a academia e a iniciativa privada que tem por objetivo fomentar a inovação e o empreendedorismo. Seus principais temas de pesquisa incluem: sistemas multiprocessados em chip (Multiprocessor System on Chip, MPSoC), projeto em nível de sistema e redes em chip (Network on Chip, NoC).

Ementa da disciplina

Estudo sobre conceitos de Classes (atributos, métodos, propriedades, visibilidade, instancia ou classe). Estudo de conceitos de Herança, Polimorfismo, Interfaces, Genéricos e Arrow functions. Estudo sobre funções de filtragem, mapeamento e redução. Estudo sobre construtores de tipos.

Organização da aula

- Introdução
- Funções
- Trabalhando com classes
- Exceções
- Funções assíncronas
- Considerações finais

Organização da aula

- Introdução
 - Contextualização
 - O ambiente de trabalho
 - Tipos de dados e manipulações fundamentais
- Funções
- Trabalhando com classes
- Exceções
- Funções assíncronas
- Considerações finais

Organização da aula

- Introdução
- Funções
 - Conceitos gerais
 - Arrow function
 - Funções de mais alta ordem
- Trabalhando com classes
- Exceções
- Funções assíncronas
- Considerações finais

Organização da aula

- Introdução
- Funções
- Trabalhando com classes
 - Modularização
 - Exemplos de uso
 - Desestruturação
 - JSON
- Exceções
- Funções assíncronas
- Considerações finais

Organização da aula

- Introdução
- Funções
- Trabalhando com classes
- Exceções
 - Conceitos
 - Exemplos de uso
- Funções assíncronas
- Considerações finais

Organização da aula

- Introdução
- Funções
- Trabalhando com classes
- Exceções
- Funções assíncronas
 - Callback, promises, async/await
- Considerações finais

Introdução

- Funções
- Trabalhando com classes
- Exceções
- Funções assíncronas
- Considerações finais

Introdução

- Contextualização
- Orientação a objeto
 - Paradigma de programação
 - Define uma regra/estilo/estrutura de codificação
 - Suportado por diferentes tecnologias/linguagens
 - Javascript, Java, C#, Python,
 - Baseada em “pilares”
 - Abstração, encapsulamento, Herança e Polimorfismo
 - Largamente empregado na indústria



Introdução

- Contextualização
 - Foco do desenvolvimento desta aula
 - Exploração de recursos de apoio da linguagem
 - Exploração do uso de classes
 - Exploração de funções assíncronas

Conceitos básicos da linguagem

- Noções básicas
 - O que é JS
 - Referência
 - https://developer.mozilla.org/pt-BR/docs/Learn/Getting_started_with_the_web/JavaScript_basics
 - Linguagem baseada em objetos
 - Dinâmica
 - Fracamente tipada
 - Usualmente interpretada por navegadores
 - Organização
 - Pode-se
 - Criar scripts que manipulam o HTML e CSS (frontend)
 - Criar scripts para serem executados no server side (backend)
 - A extensão do arquivo de script é usualmente .js



.js

Questão filosófica

- Vale a pena explorar OO com Javascript?
 - Porque da pergunta?
 - Javascript é uma linguagem “muito flexível”
 - Não requer uso de *class* para definir objetos, como java
 - Não dá suporte claro a recursos tais com o genérico e interfaces
 - etc
 - A resposta é sim, pois
 - Contribui sobremaneira na construção de soluções
 - Outras tecnologias (bibliotecas/frameworks) correlatas exploram isso muito bem
 - Mantemos o foco em uma tecnologia em constante manutenção
 - ECMAScript tem sua revisão constante com atualização recente e comunidade bastante ativa



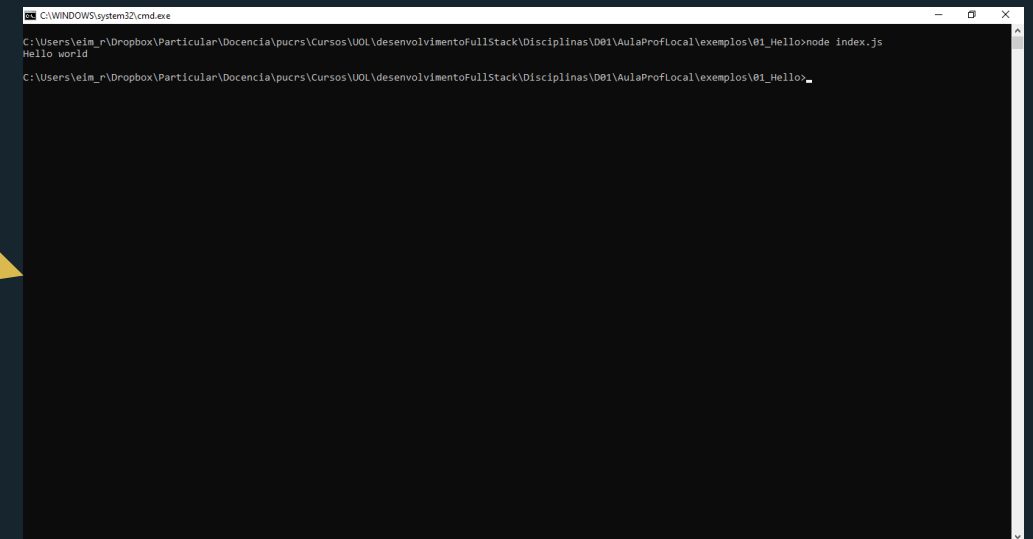
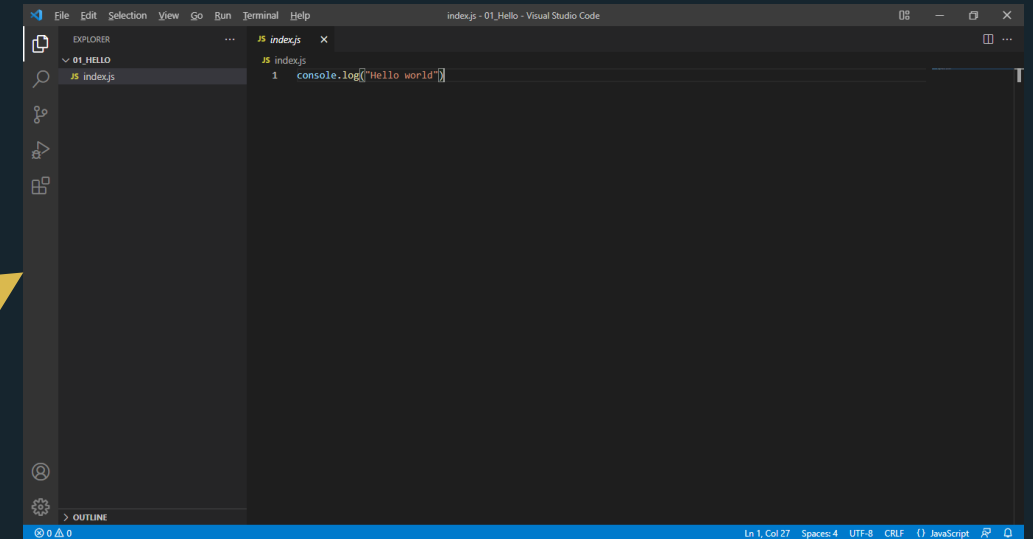
Ambiente de desenvolvimento

- Soluções online
 - Reduzem a complexidade de instalações
 - Garantem ambiente estável
 - Normalmente possuem exemplos base
- Algumas propostas
 - <https://repl.it/languages/javascript>
 - <https://stackblitz.com/fork/js>
 - <https://ronakshah.net/console/>
 - <https://es6console.com/>



Ambiente de desenvolvimento

- Proposta de ambiente local
 - O que será nesta aula
 - Ambiente de edição dos arquivos
 - VSCode
 - <https://code.visualstudio.com/>
 - Ambiente de execução
 - Node
 - <https://nodejs.org/en/download/>
 - Validando o ambiente
 - Criar um projeto hello world
 - `console.log("Hello world");`



var, let e const

- var
 - Declara uma variável com escopo de função
 - Pode ter o valor de inicialização definido
- let
 - Declara uma variável com escopo de bloco ou variável local
 - Pode ter o valor de inicialização definido
- const
 - Declara uma constante com escopo de bloco
 - Atribuição única mas com valores de objeto alterável

var, let e const

- Exemplos - VAR
 - As variáveis var01 e var02 são acessíveis sem problemas
 - Código do meio tem uma peculiaridade (undefined)

```
//01a_var.js
function testVar(){
  var var01="var 01"
  {
    var var02="var 02"
    console.log("01a."+var01)
    console.log("01b."+var02)
  }
  console.log("02a."+var01)
  console.log("02b."+var02)
}

testVar()
```

```
//01b_var.js
function testVar(){
  var var01="var 01"
  if(1==0){
    var var02="var 02"
    console.log("01a."+var01)
    console.log("01b."+var02)
  }
  console.log("02a."+var01)
  console.log("02b."+var02)
}

testVar()
```

```
//01c_var.js
function testVar(){
  var var01="var 01"
  {
    console.log("01a."+var01)
    var var01="var 02"
    console.log("02a."+var01)
  }
  var var01="var 03"
  console.log("03a."+var01)
}

testVar()
```

var, let e const

- Exemplos - LET
 - As variáveis são acessadas normalmente

```
//02a_let.js
function testVar(){
  let let01="let 01"
  {
    let let02="let 02"
    console.log("01a."+let01)
    console.log("01b."+let02)
  }
  console.log("02a."+let01)
}

testVar()
```

var, let e const

- Exemplos - LET
 - A variável let02 não é acessível fora do bloco de definição
 - Haverá erro no comando: `console.log("02b."+let02)`

```
//02b_let.js
function testVar(){
  let let01="let 01"
  {
    let let02="let 02"
    console.log("01a."+let01)
    console.log("01b."+let02)
  }
  console.log("02a."+let01)
  console.log("02b."+let02)
}

testVar()
```

var, let e const

- Exemplos - LET

- A redefinição da variável let01 não é suportada
 - Haverá erro no comando: console.log("01a."+let01)
 - A variável let01 causará conflito pois pode ser a primeira ou a segunda declaração

```
//02c_let.js
function testVar(){
    let let01="let 01"
    {
        console.log("01a."+let01)
        let let01="let 02"
        console.log("02a."+let01)
    }
    console.log("03a."+let01)
}

testVar()
```

var, let e const

- Exemplos - CONST
 - O código exemplo terá dois erros
 - const01 não pode receber novo valor
 - Haverá erro no comando: `const01 = "novo valor"`
 - const02 não é acessíveis fora do bloco
 - Haverá erro no comando: `console.log("02b."+let02)`

```
//03a_const.js
function testConst(){
  const const01="const 01"
  {
    const const02="const 02"
    console.log("01a."+const01)
    console.log("01b."+const02)
  }
  const01 = "novo valor"
  console.log("02a."+const01)
  console.log("02b."+const02)
}

testConst()
```

var, let e const

- Exemplos - CONST
 - O código da esquerda roda sem erro
 - O código da direita apresenta um erro
 - Redefinição do objeto não é permitida
 - Erro no comando: `const01 = {propriedade: "novo valor"}`

```
//03b_const.js
function testConst(){
    const const01={propriedade: "valor"}
    console.log("01a."+const01.propriedade)

    const01.propriedade = "novo valor"
    console.log("02a."+const01.propriedade)
}

testConst()
```

```
//03c_const.js
function testConst(){
    const const01={propriedade: "valor"}
    console.log("01a."+const01.propriedade)

    const01 = {propriedade: "novo valor"}
    console.log("02a."+const01.propriedade)
}

testConst()
```

Objetos

- Conjuntos diferentes de objetos são disponibilizados:
 - Intrínsecos ao JavaScript
 - Array, Boolean, Date, Math, Number, String, RegExp, Global
 - Fornecidos pelo navegador
 - Window, Navigator, Screen, History, Location, Console
 - Fornecidos pela API DOM
 - Document, Event, etc
- Cada objeto possui métodos e propriedades

Tipos de Dados

- Undefined (primitivo)
 - Valor único *undefined*, representa um valor de variáveis que não receberam nenhuma atribuição, que não foi inicializada e que não pode nem ter seu tipo definido
- Null (primitivo)
 - Valor único *null*, representa a ideia de “nada” ou “vazio”
- Boolean (primitivo)
 - Valores *true* ou *false*
- String (primitivo)
 - Valores de sequência de caracteres
- Number (primitivo)
 - Valores numéricos
- Object
 - Representa o conceito de objeto na linguagem
- Symbol (primitivo)
 - Representa identificadores únicos para objetos



Number

- Valores numéricos de ponto flutuante 64 bits padrão IEEE754
 - 64 bit de precisão dupla IEEE 754
 - Valores especiais *NaN*, *+Infinity* e *-Infinity*
- Valores
 - Por padrão em decimal (34)
 - hexadecimal inicial por 0x (0x34)
 - octal iniciam por 0o (0o34)
 - binário iniciam por 0b (0b00110100)
- Propriedades:
 - *MAX_VALUE*, *MIN_VALUE*, etc
- Métodos:
 - *toExponential()*, *toFixed()*, *toPrecision()*, *toString()*, *valueOf()*

String

- Sequência imutável de caracteres Unicode UTF-16
 - Representada por caracteres entre `""` ou `''`
- Permite interpolação
 - Quando representadas entre ```, permitem embutir valores de variáveis ou expressões via `${ }`
 - Exemplo: ``Hello ${nome}``
- Indexável
 - É possível acessar caracteres por posição (inicia em 0) via `[]`
- Propriedades:
 - *length* – informa o número de caracteres
- Métodos:
 - *charAt()*, *indexOf()*, *split()*, *substring()*, *toUpperCase()*, etc

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

Math

- Objeto que possui a definição de constantes e operações matemáticas de uso geral
- Propriedades:
 - *E*, *PI*, *LN2*, etc
- Métodos:
 - *abs()*, *sin()*, *exp()*, *max()*, *pow()*, *random()*, etc

Date

- Suporta a manipulação de tempo e data

- Construtor:

- Ano possui 4 dígitos
 - Mês de 0 a 11

```
let hoje = new Date();  
let dia = new Date(2017, 4, 2);
```

- Comparação:

- Suporta comparação via >, <, etc

```
hoje < d
```

- Métodos:

- getFullYear(), getMonth(), getDate(), getDay(), getHours(), getMinutes(), getSeconds(), getMilliseconds(), getTime(), toString(), toDateString(), setFullYear(), setMonth(), setDate(), setHours(), setMinutes(), setSeconds(), setMilliseconds(), setTime(), etc

Global

- Objeto que possui várias propriedades e métodos de uso geral
 - Em um navegador, recebe o nome de *window*
 - No NodeJS, recebe o nome de *global*
- Propriedades:
 - Infinity, NaN, undefined
- Métodos:
 - parseFloat(string) e parseInt(string) – convertem uma string para número
 - escape(string) e unescape(string) – codifica/decodifica uma string
 - eval(string) – avalia e executa o conteúdo da string com código de script
 - etc

Conversão entre Tipos

- Operadores e funções normalmente realizam a conversão automática
 - Para converter explicitamente para *string*, utiliza-se a função `String(valor)`
 - Para converter explicitamente para *Number*, utiliza-se a função `Number(valor)`
 - `undefined` resulta em `NaN`
 - `null` resulta em `0`
 - `true` resulta em `1`
 - `false` resulta em `0`
 - `string` resulta no valor numérico ou `NaN` caso não represente literal numérico

Arrays

- Um objeto do tipo Array
 - Permite o armazenamento de uma coleção de múltiplos itens
 - Possui propriedades e comportamento
 - Permite executar operações comuns em arrays
- Armazenamento baseado em índice
- Funcionam como base para estruturas de dados
 - listas, filas e pilhas

Arrays

- Declaração de arrays:

- Literal

```
let a = [];
```

```
let a = ["A", "B", "C"];  
console.log(a);
```

- Indexação de elementos:

- Começa no índice 0

```
let c = a[1];
```

- Operador []

```
a[2] = "c";
```

Arrays

- O comprimento do array definido na propriedade `length`

```
console.log(a.length);
```

- Comprimento do array poder ser aumentado atribuindo-se um valor a uma posição de índice maior ao tamanho atual

```
a[3] = "D";
```

- Os array são esparsos
 - Somente reservam espaço para os elementos definidos
 - Aumentar o tamanho do array gera posições intermediárias com valores indefinidos!
 - Usar `push` para inclusão na última posição livre

Arrays

- Crescimento de arrays e reserva de espaços

```
//01a_length.js
var a = []
a[0] = Math.random()
console.log(a.length)
for(let val in a)
    console.log(" --> "+val)

console.log(" ----- ")

a.push(Math.random())
console.log(a.length)
for(let val in a)
    console.log(" --> "+val)
```

```
//01a_length.js
// ... continuação
console.log(" ----- ")
a[9] = Math.random()
console.log(a.length)
for(let val in a)
    console.log(" --> "+val)

console.log(" ----- ")

for(let i=0; i<a.length; i++)
    console.log(" -->"+i+": "+a[i])
```

Arrays - Iteração

- Comando de repetição do tipo for

```
//02a_loop.js
var a = []
a[0] = Math.random()
a[1] = Math.random()
a[9] = Math.random()
console.log(a.length)
```

```
//02a_loop.js
// ... Primeiro for
console.log(" -for .. in- ")
console.log(" ===== ")
for(let val in a)
    console.log(" --> "+val)
console.log(" ===== \n")
```

```
//02a_loop.js
// ... Segundo for
console.log(" -for .. of- ")
console.log(" ===== ")
for(let val of a)
    console.log(" --> "+val)
console.log(" ===== \n")
```

```
//02a_loop.js
// ... Terceiro for

console.log(" -for (;;)- ")
console.log(" ===== ")
for(let i=0; i<a.length; i++)
    console.log(" -->"+i+": "+a[i])
```

Arrays - Métodos

- Métodos:

- toString() – retorna uma string com os valores do array separados por vírgula

```
console.log(a.toString());
```

- join() – retorna uma string com os valores do array separados pelo símbolo fornecido

```
console.log(a.join(" - "));
```

- concat() – retorna um novo array resultante da concatenação dos arrays passados por parâmetro

```
let a2 = a.concat(["X", "Y"]);
```

- slice(indice, fim) – particiona um array e retorna um novo array com a partição, sem alterar o array original

```
let a = ["A", "B", "C"];  
let a3 = a.slice(1); //a[1], a[2]  
let a4 = a.slice(0,2); //a[0], a[1]
```

Arrays - Métodos

- Métodos:
 - `indexOf(item, inicio)` – retorna o índice do array que contem o elemento *item*, opcionalmente a partir da posição *inicio*; ou -1 caso não encontre
 - `lastIndexOf(item, inicio)` – retorna o índice da última ocorrência do elemento *item* no array (ou seja, realiza a busca de trás para frente), opcionalmente a partir da posição *inicio*; ou -1 caso não encontre
 - `includes(item, inicio)` – retorna `true` caso o array contenha o elemento *item*, opcionalmente a partir da posição *inicio*; ou `false` caso contrário

```
let a = [1,2,2];  
console.log(a.indexOf(2));  
console.log(a.indexOf(2,2));  
console.log(a.lastIndexOf(2));  
console.log(a.includes(0));
```

Arrays - Métodos

- Métodos:
 - `findIndex(funçãoPredicado)` – retorna o index do primeiro elemento do array de acordo com a função de predicado; retorna -1 caso contrário
 - A função de predicado é uma função que retorna true ou false
 - A função de predicado tem a assinatura `function(item,index,array)`, onde *item* é o elemento, *index* é a posição atual do elemento, *array* é o próprio array; usualmente utiliza-se somente o primeiro parâmetro
 - `find(funçãoPredicado)` – retorna a primeira ocorrência de um elemento no array de acordo com a função de predicado: retorna *undefined* caso contrário

```
let a = [1,2,3];
let i = a.findIndex(function(item) {
    return item >= 2;
});
let e = a.findIndex(function(item) {
    return item >= 0 && item <= 2;
});
```

Arrays - Métodos

- Métodos:
 - `forEach(funçãoAplicação)` – itera sobre cada elemento do array e chama a função de aplicação sobre cada um
 - A função de aplicação tem a assinatura `function(item,index,array)`, onde *item* é o elemento, *index* é a posição atual do elemento, *array* é o próprio array; usualmente utiliza-se somente o primeiro parâmetro

```
let a = [1,2,3];
a.forEach(function(item,index) {
    console.log(`${item} na posição ${index}`);
});
```


Arrays - Ordenação

- Métodos:

- `sort()` – ordena um array de forma ascendente, de acordo com o tipo string (ordem lexicográfica)

```
let a = ["A", "B", "C"];  
a.sort();
```

- `reverse()` – ordena um array de forma descendente, de acordo com o tipo string (ordem lexicográfica)

```
let a = ["A", "B", "C"];  
a.reverse();
```

Arrays - Ordenação

- Métodos:
 - `sort(funçãoDeComparação)` – ordena um array de acordo com a função de comparação fornecida
 - A função de comparação deve comparar dois valores e retornar um número negativo (primeiro menor que segundo), número positivo (primeiro maior que segundo), zero (caso contrário)

```
let a = [3, 1, 2];  
a.sort(function(x,y) {  
    if (x<y) return -1;  
    if (x>y) return 1;  
    return 0;  
});
```

- Introdução

Funções

- Trabalhando com classes
- Exceções
- Funções assíncronas
- Considerações finais

Funções

- Funções
 - Definidas com a palavra reservada *function*
 - Pode possuir um nome
 - Existe o conceito de função anônima
 - Pode ter argumentos e retornar valor
 - A passagem de parâmetros é por valor
 - O número de parâmetros passados não é verificado
 - Se não recebe um valor, o parâmetro tem valor undefined

Funções

- Exemplos
 - Funções que não recebem parâmetros

```
// 01a_funcaoSemParametro
function funcaoSemRetorno() {
    console.log("Alô Mundo!");
}

function funcaoComReturn() {
    return "Alô Mundo!";
}

let msg = funcaoSemRetorno()
console.log(msg)

msg = funcaoComReturn()
console.log(msg)
```

Funções

- Exemplos
 - Funções que recebem parâmetros

```
// 01b_funcaoComParametro
function potencia(base, expoente = 2) {
    let resultado = 1;
    for (let cont = 0; cont < expoente; cont++) {
        resultado *= base;
    }
    return resultado;
}

console.log(potencia());
console.log(potencia(4));
console.log(potencia(2,6));
console.log(potencia(2,6,18));
```

Além da função básica

- JavaScript não é uma linguagem de programação funcional, mas...
 - Permite manipular funções como objetos, logo....
 - Pode-se usar técnicas de programação funcional em JS
- Métodos de array do The ECMAScript 5 tais como `map()` e `reduce()`
 - Prestam-se a tal estilo de programação funcional

Além da função básica

- Em JavaScript, funções podem ser manipuladas assim como valores
 - Pode-se atribuir a variáveis,
 - Pode-se passar a função como parâmetro para outra função
 - Pode-se retornar uma função como valor de retorno de outra função, etc
- Uma forma alternativa de definir funções é através de expressões:

```
let identificador = function (lista de parâmetros) { bloco de comandos };
```

- Outra forma alternativa de definir funções é através de “expressões lambda”:

```
let identificador = (lista de parâmetros) => expressão;
```

```
let identificador = (lista de parâmetros) => {bloco de comandos};
```


Funções Manipuláveis

Exemplo – função como argumento

```
// 02a_funcaoComoParametro
function decision(question, doOK, doCancel) {
    if (question=="OK") doOK()
    else doCancel();
}

function showOk() { console.log( "You agreed." ); }

function showCancel() { console.log( "You canceled the execution." ); }

decision("OK", showOk, showCancel);
decision("Cancel", showOk, showCancel);
```

Funções Manipuláveis

Exemplo – função anônima como argumento

```
// 02b_funcaoComoParametro
function decision(question, doOK, doCancel) {
    if (question=="OK") doOK()
    else doCancel();
}

decision(
    "OK",
    function () { console.log( "You agreed." ); },
    function () { console.log( "You canceled the execution." ); }
);

decision(
    "Cancel",
    function () { console.log( "You agreed." ); },
    function () { console.log( "You canceled the execution." ); }
);
```

Funções Manipuláveis

Exemplo – função atribuída a variável como argumento

```
// 02c_funcaoComoParametro
function decision(question, doOK, doCancel) {
    if (question=="OK") doOK()
    else doCancel();
}

let beOK = function showOk() { console.log( "You agreed." ); }

let beCancel = function showCancel() { console.log( "You canceled the execution." ); }

decision("OK", beOK, beCancel);
decision("Cancel", beOK, beCancel);
```

Funções de fechamento (closure)

- Closure
 - Função que se "lembra" do ambiente em que ela foi criada
 - Permite associar dados do ambiente com uma função que trabalha estes dados.
 - Execução com contexto
 - Diretamente ligado com programação orientada a objetos
 - Objetos nos permitem associar dados utilizando um ou mais métodos

```
// 03a_closure
function somaValores(x) {
  return function(y) {
    return x + y;
  };
}

var soma5 = somaValores(5);

console.log(soma5(2));
```

Funções de seta (arrow functions)

- Sintaxe mais curta quando comparada a uma função
- Restrições
 - Melhora aplicadas para funções que não sejam métodos
 - Não podem ser usadas como construtoras de objetos.

```
//SINTAXE BÁSICA
(param1, param2, ..., paramN) => { statements }
(param1, param2, ..., paramN) => expression
// equivalente a: => { return expression; }

// Parênteses são opcionais quando só há um nome de parâmetro:
(singleParam) => { statements }
singleParam => { statements }

// A lista de parâmetros para uma função sem parâmetros
// DEVE ser escrita com um par de parênteses ou _.
() => { statements }
_ => { statements }
```

Funções de seta (arrow functions)

- Exemplos de construção de arrow functions

```
// 04a_arrow
var somar = () => console.log("função sem parametros")
somar()
somar(1)

somar = _ => console.log("usando underscore")
somar()

somar = (x,y) => x + y;
console.log(somar(1,2));

somar = (x,y) => {return x + y};
console.log(somar(3,4));
```

```
// 04a_arrow
// ... continuação

// retorna o maior numero
somar = (x,y) => x>y?x:y;
console.log(somar(5,6));

somar = (x,y) => {
    if(x>y)
        return x
    else
        return y
}
console.log(somar(7,8));
```

Funções de alta ordem

- Explorar funções de transformação sobre arrays com arrow functions
 - Métodos de Array a serem explorados
 - `Array.some()`
 - Sinaliza se ao menos um dos elementos do array atende a regra
 - `Array.every()`
 - Sinaliza se todos os elementos do array atendem a regra
 - `Array.filter()`
 - Retorna um novo array com a lista de itens que atende a regra
 - `Array.forEach()`
 - Itera sobre cada um dos itens da lista, aplicando alguma ação
 - `Array.reduce()`
 - Acumula itens do array conforme uma regra
 - `Array.map()`
 - Permite transformar os elementos da lista

Funções de alta ordem

- Array.some()

```
// 05a_arraySome
array = [4,5,6,7,8,9,10]

regraImpar = (item) => item%2 == 0

console.log("Há algum número impar? "+
array.some(regraImpar))
```

```
// 05a_arraySome
// ... continuação
regraPrimo = (item) => {
    let ndiv=0;
    for(let divisor=1; divisor<=item; divisor++)
        if((item%divisor)== 0)
            ndiv++

    if(ndiv==2)
        return true
    else
        return false
}

console.log("Há algum número primo? "+
array.some(regraPrimo))
```


Funções de alta ordem

- Array.every()

```
// 05b_arrayEvery
array = [4,5,6,7,8,9,10]

regraImpar = (item) => item%2 !== 0

console.log("Todos os números que são impar? "+
array.every(regraImpar))
```

```
// 05b_arrayEvery
// ... continuação
regraPrimo = (item) => {
    let ndiv=0;
    for(let divisor=1; divisor<=item; divisor++)
        if((item%divisor)== 0)
            ndiv++

    if(ndiv==2)
        return true
    else
        return false
}

console.log("Todos os números que são primos? "+
array.every(regraPrimo))
```

Funções de alta ordem

- Array.filter()

```
// 05c_arrayFilter
array = [4,5,6,7,8,9,10]

regraImpar = (item) => item%2 !== 0

console.log("Filtrar números impar? "+
array.filter(regraImpar))
```

```
// 05c_arrayFilter
// ... continuação
regraPrimo = (item) => {
    let ndiv=0;
    for(let divisor=1; divisor<=item; divisor++)
        if((item%divisor)== 0)
            ndiv++

    if(ndiv==2)
        return true
    else
        return false
}

console.log("Filtrar os números primos? "+
array.filter(regraPrimo))
```

Funções de alta ordem

- `Array.forEach()`

```
// 05d_arrayForEach
array = [4,5,6,7,8,9,10]

array.forEach(
  (nro) => console.log(
    nro+
    " -> "+
    (nro%2==0?"par":"ímpar"))
)

nroDivisores = (item) => {
  let ndiv=0;
  for(let divisor=1; divisor<=item; divisor++)
    if((item%divisor)== 0)
      ndiv++
  return ndiv;
}
```

```
// 05d_arrayForEach
// ... continuação

array.forEach(
  (nro) => console.log(
    nro+
    `-> nDivisores de 1 até ${nro} =`+
    nroDivisores(nro) )
)
```

Funções de alta ordem

- `Array.reduce()`

```
// 05e_arrayReduce
array = [4,5,6,7,8,9,10]

let resultado =
  array.reduce(
    (acc, val) => acc+=(val%2==0)?val:0,
    0
  )

console.log("A soma dos nros pares é "+resultado)
```

Funções de alta ordem

- Array.map()

```
// 05f_arrayMap
array = [4,5,6,7,8,9,10]

var newArray = array.map( (item) => item*2 )
console.log(newArray)

newArray = array.map( (item) => {return {x: item, y: 2*item}} )
console.log(newArray)
```

Funções de alta ordem

- Juntando Operações

```
// 05g_arrayFilterMapForEach
array = [2,3,4,5,6,7,8,9,10]

nroDivisores = (item) => {
  let ndiv=0;
  for(let divisor=1; divisor<=item; divisor++)
    if((item%divisor)== 0)
      ndiv++
  return ndiv;
}

array
  .filter((nro) => nroDivisores(nro)==2)
  .map((item) => {return {x:item, par:(item%2)==0}} )
  .forEach((obj) => console.log(obj.x + " é par? " + obj.par))
```

- Introdução
- Funções

Trabalhando com classes

- Exceções
- Funções assíncronas
- Considerações finais

Modularização

- 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 6 Modules
 - etc

Módulo - CommonJS

- Característica
 - Padrão utilizado por um grande número de pacotes disponibilizados via NPM
 - Ambiente de execução do NodeJS suporta o padrão CommonJS
- Módulos que contém as definições
 - Definem suas interfaces via `exports` e `module.exports`
 - Usa-se **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 função `require`

Módulo - CommonJS

- Definição do módulo (01a_definicao.js):
 - Exporta funções no objeto padrão

```
//01a_definicaoCJS.js
exports.area = (r) => Math.PI * r**2;
exports.circunferencia = (r) => 2 * Math.PI * r;
```

- Importando o módulo (01a_consumidor.js):

```
//01a_consumidorCJS.js
const circulo = require('./01a_definicaoCJS.js');
console.log(`Área do círculo de raio 4 é ${circulo.area(4)}`);

//desestruturando o objeto e acessando a função diretamente
const {area} = require('./01a_definicaoCJS.js');
console.log(`Área do círculo de raio 2 é ${area(2)}`);
```

- Rodar com o comando: `node 01a_consumidor.js`

Módulo - CommonJS

- Definição do módulo (02a_definicao.js): exportando objeto

```
//02a_definicaoCJS.js - ALTERNATIVA A
class Circulo {
  constructor(r) {
    this.raio = r;
  }
  area() {
    return Math.PI * this.raio**2;
  }
  circunferencia() {
    return 2 * Math.PI * this.raio;
  }
};
```

```
//02a_definicaoCJS.js - ALTERNATIVA B
module.exports = class Circulo {
  constructor(r) {
    this.raio = r;
  }
  area() {
    return Math.PI * this.raio**2;
  }
  circunferencia() {
    return 2 * Math.PI * this.raio;
  }
};

module.exports = Circulo
```

Módulo - CommonJS

- Importando o módulo:

```
//02a_consumidorCJS.js
const Circulo = require('./02a_definicaoCJS.js');

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

Módulos – ES6

- Padrão nativo do JavaScript disponível a partir do ECMAScript 6 (2015)
- Ambiente de execução do NodeJS suporta o padrão ES6
 - Flag `--experimental-modules` ao executar Node
- Módulos definem suas interfaces via palavra-chave `export`
 - Pode-se exportar múltiplas funções, classes, `let`, `const`, `var`
 - Vinculação de exportação default é tratado como elemento principal do módulo
- 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`

Módulo – ES6 no NodeJS

- Definição do módulo (03a_definicao_ES6.mjs):
 - Exportando funções no objeto padrão

```
//03a_definicao_ES6.mjs
export function area(r) { return Math.PI * r ** 2; }
export function circunferencia(r) { return 2 * Math.PI * r; }
```

- Importando o módulo (03a_definicao_ES6.mjs):

```
//03a_consumidor_es6.mjs
import {area, circunferencia as circ} from "./03a_definicao_ES6.mjs";
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 "./03a_definicao_ES6.mjs";
console.log(`Área do círculo de raio 2 é ${circulo.area(2)}`);
```

Módulo – ES6 no NodeJS

- Definição do módulo (04a_definicao_ES6.mjs): exportando objeto

```
//04a_definicao_ES6.mjs
export class Circulo {
  constructor(r) {
    this.raio = r;
  }
  area() {
    return Math.PI * this.raio ** 2;
  }
  circunferencia() {
    return 2 * Math.PI * this.raio;
  }
}
```

Módulo – ES6 no NodeJS

- Importando o módulo (04a_consumidor_ES6.mjs):

```
//04a_consumidor_es6.mjs
import {Circulo} from "./04a_definicao_ES6.mjs";
const c1 = new Circulo(4);
console.log(`Área do círculo de raio 4 é ${c1.area()}`);
```


Casos de exemplo de construção – CASO 01

- Classe carro com atributo privado, getter e setter
 - Razões da privacidade
 - Alternativa para getter e setter

```
//05a_carro.mjs
export class Carro{
  #tanque
  #capacidadeDoTanque
  constructor(valor){
    this.#tanque=0
    this.#capacidadeDoTanque=valor
  }
  get tanque(){
    return this.#tanque
  }
  get capacidade(){
    return this.#capacidadeDoTanque
  }
}
```

```
//05a_carro.mjs
// ... continuação
  set tanque(qtde){
    if(qtde>=0)
      if(qtde+this.#tanque > this.#capacidadeDoTanque)
        this.#tanque=this.#capacidadeDoTanque;
      else
        this.#tanque += qtde
    }
  }
```

Casos de exemplo de construção – CASO 01

- Manipulação da classe Carro

```
//05a_consumidor.mjs
import {Carro} from
"./05a_carro.mjs"

var carro = new Carro(55)
console.log(carro.capacidade)
console.log(carro.tanque)

carro.tanque = 30
console.log(carro.tanque)

carro.tanque = 30
console.log(carro.tanque)

carro.tanque = 30
console.log(carro.tanque)
```

Casos de exemplo de construção – CASO 02

- Classe Herdada CarrosComPlaca, Composição com a classe Locadora
 - Classe CarroComPlaca que estende Carro

```
//06a_CarroComPlaca.mjs
import {Carro} from "../05_simpleClass/05a_carro.mjs"

export class CarroComPlaca extends Carro{
  #_placa
  constructor(umaPlaca){
    super(55)
    this.#_placa=umaPlaca
  }

  get placa(){
    return this.#_placa
  }
}
```

Casos de exemplo de construção – CASO 02

- Classe Herdada CarrosComPlaca, Composição com a classe Locadora
 - Classe Locadora, que explora array, funções de massa de dados e *arrow function*

```
//06a_Senha.mjs
import {CarroComPlaca} from "../06a_CarroComPlaca.mjs"

export class Locadora{

  #_carros

  constructor(){
    this.#_carros=[]
  }

  adicionaCarro(umCarro){
    this.#_carros.push(umCarro)
    console.log(this.#_carros.length)
  }
}
```

```
//06a_Senha.mjs
// ... continuação
consultaCarros(){
  this.#_carros.forEach(
    (carro) => console.log(
      "Carro placa (" +
      carro.placa + "); tq:" +
      carro.tanque )
  )

  abasteceCarro(index, quantidade){
    if(index >= 0 && index < this.#_carros.length)
      this.#_carros[index].tanque = quantidade
  }
}
```

Casos de exemplo de construção – CASO 02

- Manipulação da classe locadora

```
//06a_consumidor.mjs
import { CarroComPlaca } from "./06a_CarroComPlaca.mjs"
import { Locadora } from "./06a_Locadora.mjs"

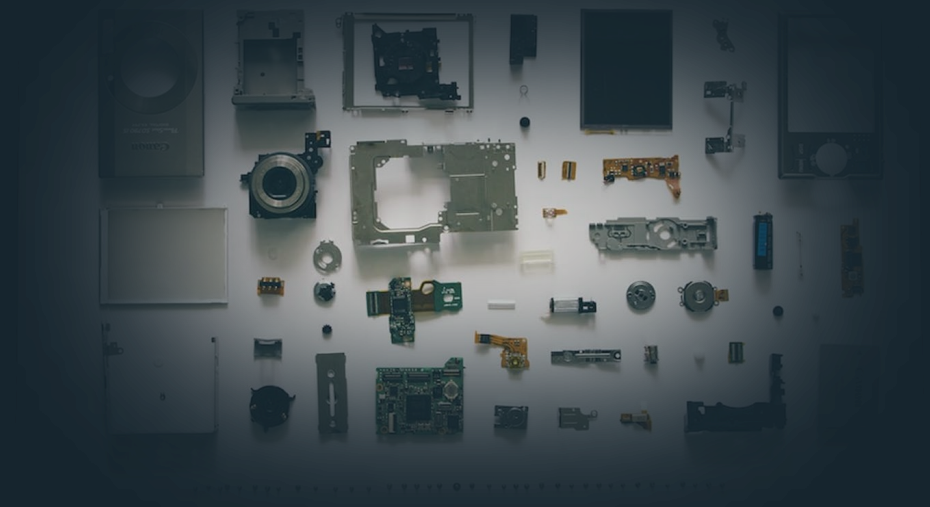
var locadora = new Locadora()
locadora.adicionaCarro(new CarroComPlaca("ABC-9I23"))
locadora.adicionaCarro(new CarroComPlaca("DEF-0U74"))
locadora.consultaCarros()

locadora.abasteceCarro(0, 30)
locadora.consultaCarros()

locadora.abasteceCarro(1, 30)
locadora.abasteceCarro(0, 30)
locadora.consultaCarros()
```

Desestruturação

- A desestruturação é um operação comum na linguagem
 - A ideia é “desempacotar” algo em vários “pedaços”
- A desestruturação é aplicável, por exemplo, em
 - Módulos importados
 - Arrays
 - Objetos
 - etc



Desestruturando Arrays → []

- A operação de atribuição pode utilizar um modo de “desestruturação” que permite funcionalidades interessantes
 - A ideia é “desempacotar” um array em vários “pedaços”

```
//01a_Array.js
let nomeCompleto = ['Edson', 'Ifarraguirre', 'Moreno'];
var [primeiroNome, nomeDoMeio, ultimoNome] = nomeCompleto;

console.log("O primeiro nome é: "+primeiroNome)
console.log("O nome do meio é: " +nomeDoMeio)
console.log("O último nome é: " +ultimoNome)

var [primeiroNome, ...restante] = nomeCompleto
console.log("O primeiro nome é: "+primeiroNome)
console.log("O restante do nome é: \"" +restante.toString().replace(',',' ' '')+'"')

var [soOPrimeiroNome, ,soOUltimoNome] = nomeCompleto
console.log("O primeiro nome é: "+soOPrimeiroNome)
console.log("O último nome é: " +soOUltimoNome)
```

Desestruturando Objetos → {}

- Objetos podem também ser desestruturados em suas partes pelo operador de atribuição
 - Ordem não importa, mapeamento é por identificador, suporta valores padrão
 - Identificador pode ser renomeado via “:”

```
//03a_umaClasse.mjs
export class UmaClasse{
  #_umAtributo
  outroAtributo = "atributo dois"
  constructor(n){
    this.#_umAtributo = n
  }
  capturaPrimeiroAtributo(){
    return this.#_umAtributo
  }
}
```

```
//03a_ObjetoDeClasse.mjs

import {UmaClasse} from "../03a_umaClasse.mjs"

let umaClasse = new UmaClasse("Primeiro atributo")

let {capturaPrimeiroAtributo: umAtributo,
    outroAtributo} = umaClasse;

console.log("um Atributo: " + umAtributo)
console.log("Outro Atributo: " + outroAtributo)
```


JSON

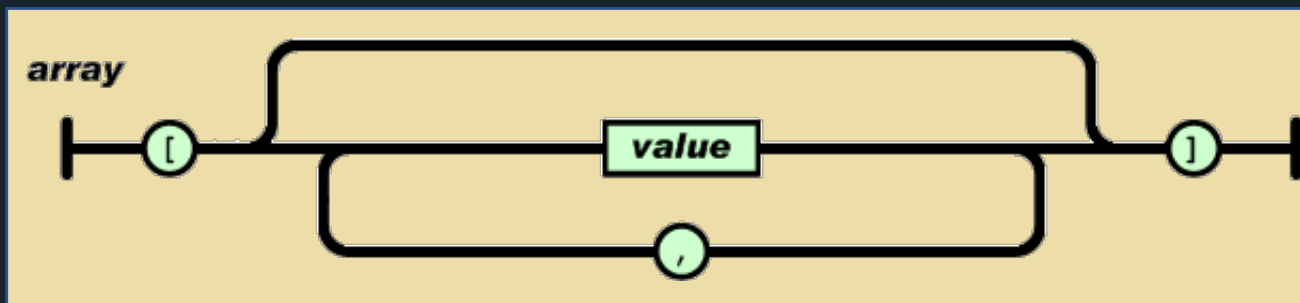
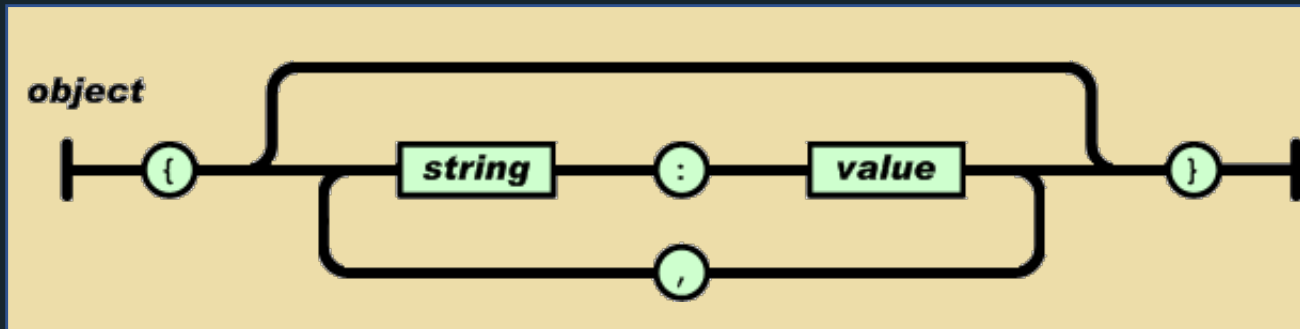
- JSON = JavaScript Object Notation
- Formato textual para serialização de dados
 - É independente de linguagem
 - Muito utilizado para retorno de Serviços Web REST

```
{  
  "productName": "Computer Monitor",  
  "price": "229.00",  
  "specifications": {  
    "size": 22,  
    "type": "LCD",  
    "colors": ["black", "red", "white"]  
  }  
}
```

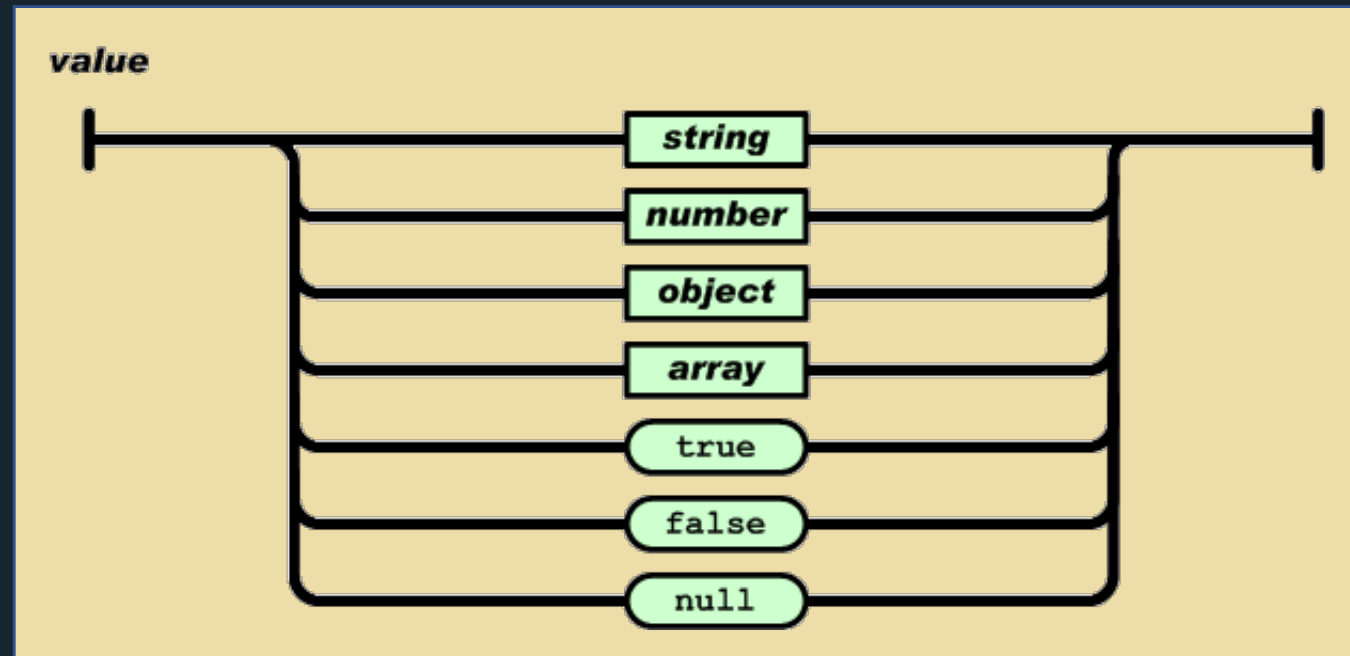
JSON

- JSON é capaz de representar:
 - Tipos primitivos
 - Strings, números, booleanos, null
 - Tipos estruturados
 - Objetos
 - Coleção não-ordenada de zero ou mais pares chave/valor
 - Arranjos
 - Coleção ordenada de zero ou mais valores

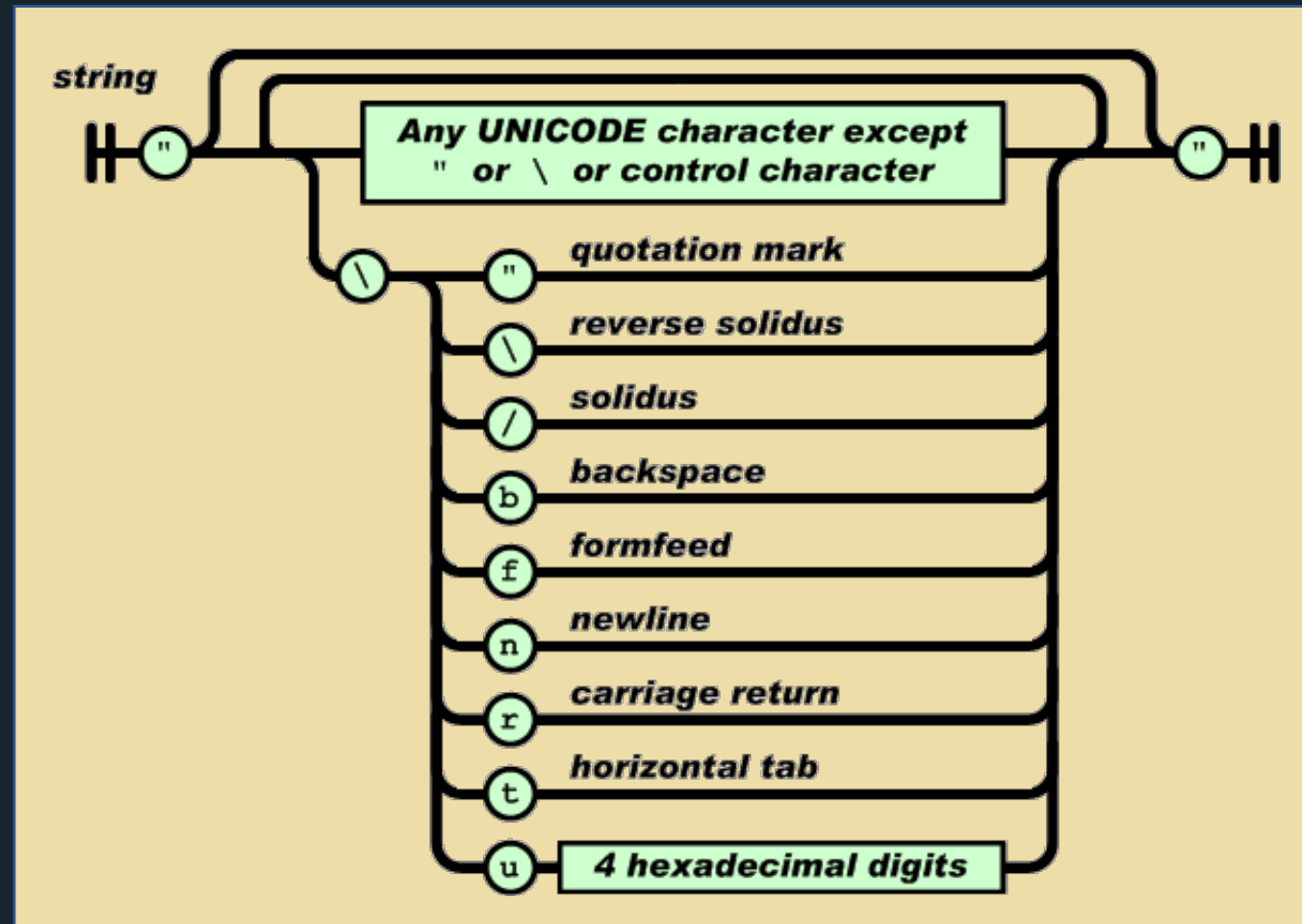
JSON



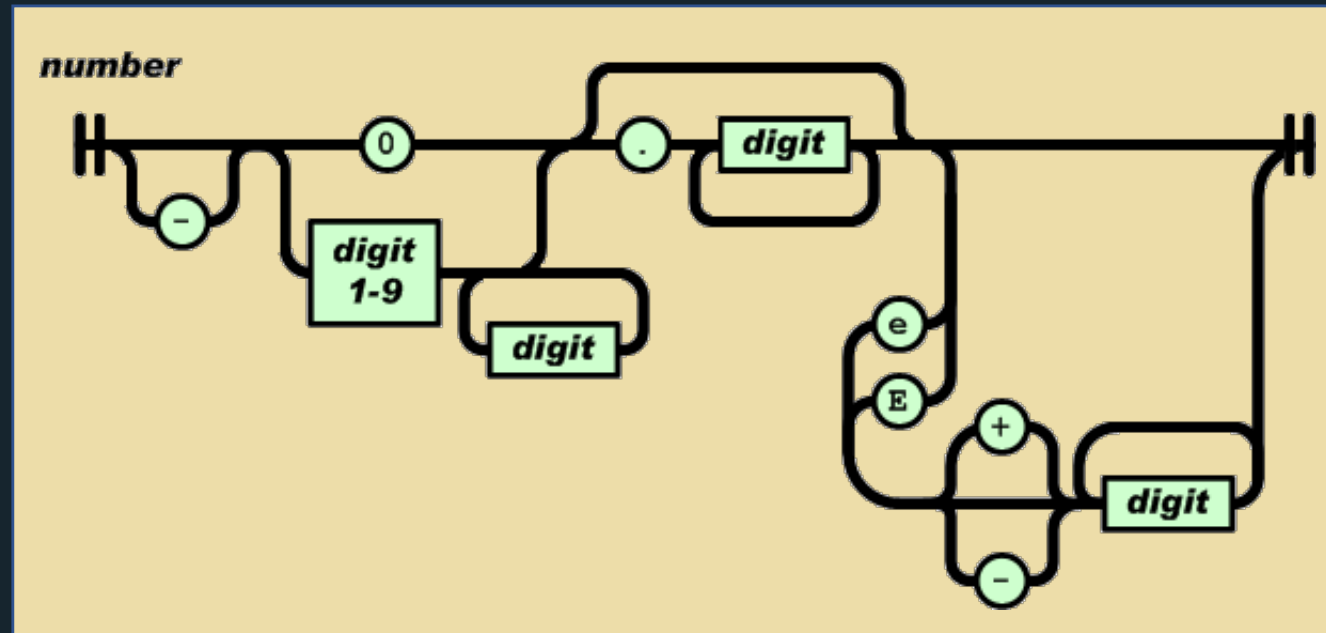
JSON



JSON



JSON



JSON

- JavaScript provê o método `JSON.stringify`
 - Converte um objeto para o formato JSON (string)
 - Cuidado: não pode existir referências circulares dentro do objeto

```
//01a_Objeto2JSON.js
let student = {
  name: 'John',
  age: 30,
  isAdmin: false,
  courses: ['html', 'css', 'js'],
  wife: null
};

let json = JSON.stringify(student);
console.log(json);
```

JSON

- JavaScript provê o método `JSON.stringify`
 - Convertendo uma instancia de uma classe com atributos privados

```
//03a_umaClasse.mjs
export class UmaClasse{
  #_umAtributo
  outroAtributo = "atributo dois"
  constructor(n){
    this.#_umAtributo = n
  }
  capturaPrimeiroAtributo(){
    return this.#_umAtributo
  }
}
```

```
//02a_ObjetoDeClasse2JSON.js

import {UmaClasse} from "../07_Deseestruturação/03a_umaClasse.mjs"

let umObjetoDeClasse = new UmaClasse("um valor")

let json = JSON.stringify(umObjetoDeClasse);

console.log(json);
```


JSON

- JavaScript provê o método `JSON.parse` para converter uma string no formato JSON em um objeto

```
//01a_Parseobjeto.js
let user = '{ "name": "John", "age": 35, "isAdmin": false, "friends": [0,1,2,3] }';
// propriedade ainda não existe, logo teremos erro
console.log(user.friends[0] ?? "Propriedade inexistente");

user = JSON.parse(user);
console.log(user.friends[0]);
```

JSON

- Cuidado! Não assuma que a conversão suporta qualquer tipo!
 - O exemplo a seguir utiliza um objeto Date, que é serializado como String e, portanto, não é desserializado automaticamente para Date
 - Deve-se utilizar a função *reviver*, que é um dos parâmetros da função JSON.parse

```
//02a_restricaoDeTipos.js
var json = '{ "name":"John Doe", "birth":"2017-11-30T12:00:00.000Z", "city":"Porto Alegre"}'

var obj = JSON.parse(json, function (key, value) {
    if (key == "birth") {
        return new Date(value);
    }
    return value;
});

console.log("Nome:  " + obj.name);
console.log("DNasc: " + obj.birth);
```

- Introdução
- Funções
- Trabalhando com classes

Exceções

- Funções assíncronas
- Considerações finais

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*
 - Qualquer valor pode ser utilizado, porém é mais adequado utilizar objetos *Error* e suas subclasses
 - Propriedades principais: *name* e *message*

```
//01a_firstException.js
try {
    throw new Error('Gerando um erro genérico!');
}
catch (e) {
    console.error(` ${e.name}: ${e.message} `);
}
```

Novas Exceções

- Criando novos tipos de exceções
 - Pode-se Herdar a classe *Error*
- Exemplo:

```
//02a_criandoExceção.js
class ValidationError extends Error {
  constructor(message) {
    super(message);
    this.name = "ValidationError";
  }
}

function vaiDarErro() {
  throw new ValidationError("Dados inválidos!");
}
```

```
//02a_criandoExceção.js
// ... continuação

try{
  vaiDarErro()
}
catch (e) {
  console.error(`${e.name}: ${e.message}`);
}
```

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 boa técnica
 - Tratar as exceções adequadas ao momento
 - Relançar as demais que não se sabe como tratar no momento

```
//03a_excecaoComParseJSON.js
let json = "incorreto"

try {
  let pessoa = JSON.parse(json);
  console.log(pessoa.nome);
}
catch(err) {
  if (err instanceof SyntaxError) {
    console.log(`Erro ${err.name}: ${err.message}`);
  } else {
    console.log("Relança a exceção pois não sabe como tratar");
    throw err;
  }
}
finally{
  console.log("Encerra tratamento")
}
```

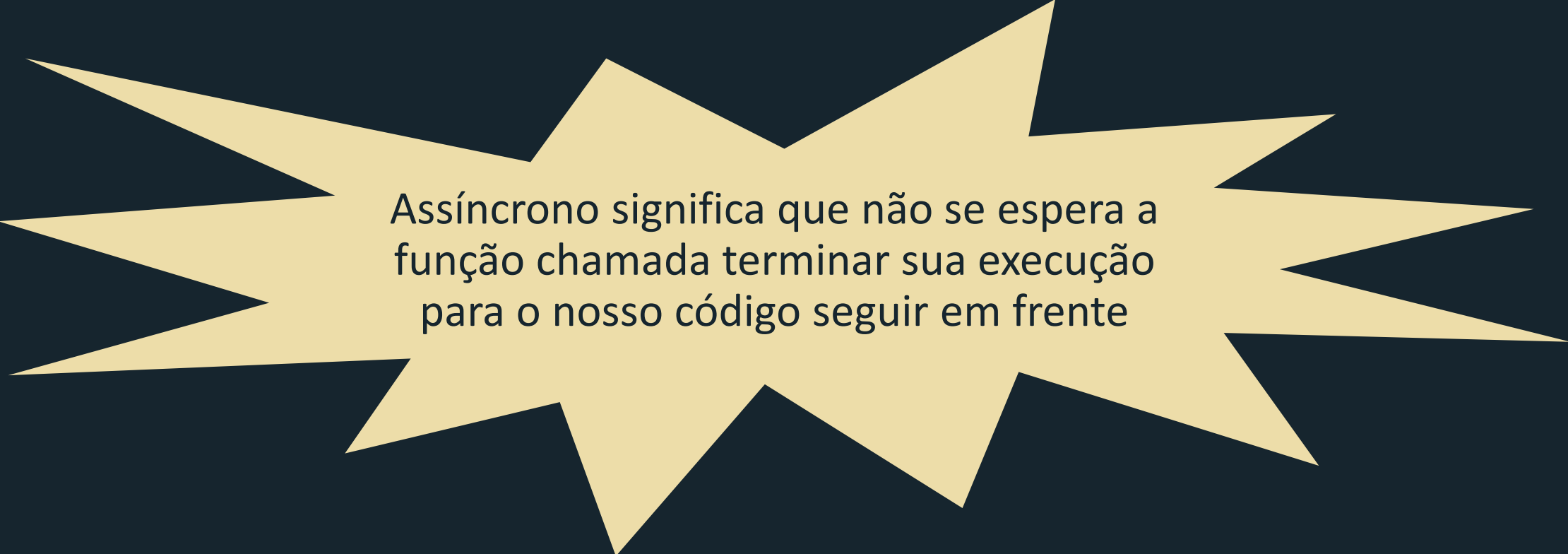

- Introdução
- Funções
- Trabalhando com classes
- Exceções

Funções assíncronas

- Considerações finais

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



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


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

```
const fs = require('fs');  
  
fs.readFile('package.json', function (err, buf) {  
    console.log(buf.toString());  
});
```

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



Callbacks - Exemplo

```
//01a_callback.js
const fs = require('fs')

const onRead = function onRead (err, buf) {
  if(err)
    console.log("houve um erro")
  else
    console.log(buf.toString())
}

fs.readFile( '01a_textoQualquer.txt', onRead )
```

```
//01b_callback.js
const fs = require('fs')

fs.readFile(
  '01a_textoQualquer.txt',
  (err, buf) => {
    if(err)
      console.log("houve um erro")
    else
      console.log(buf.toString())
  })
```

```
//01c_callback.js
const fs = require('fs')

fs.readFile(
  '01a_textoQualquer.txt',
  (err, buf) => {
    if(err) throw err
    else console.log(buf.toString())
  })
```

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 *call-backs*
- 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

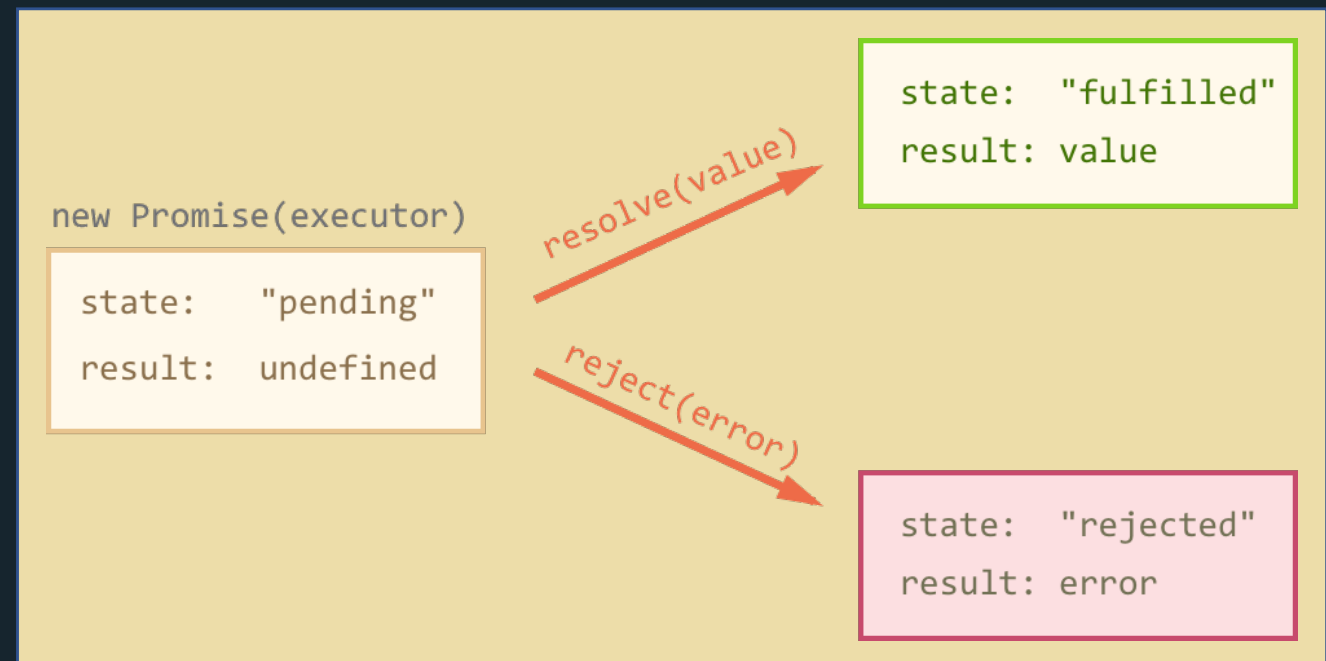
- Construtor de objetos Promise:

```
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

- 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

```
//02a_basicPromiseSucceed.js
const myFirstPromise = new Promise((resolve, reject) => {
  setTimeout(() => {
    resolve("Sucesso!");
  }, 2000);
});

let ifSucceed = (successMessage) => {
  console.log(`Finalizado! ${successMessage}`);
}

myFirstPromise
  .then(ifSucceed);

console.log("Fim do programa")
```

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)*

```
//02b_basicPromiseFail.js
const myFirstPromise = new Promise((resolve, reject) => {
  setTimeout(() => reject("Rejeitado"), 2000);
});

let ifFail = (err) => console.log(`Uma exceção foi lançada`);

myFirstPromise
  .then(
    (msg) => console.log("All righth!!"),
    (msg) => {throw "Não fui atendido!!!"} )
  .catch( ifFail )

console.log("Fim do programa")
```

Promises

- As promises podem ser encadeadas
 - Permite o a sequencialização de chamadas de funções assíncronas
- O padrão de codificação
 - A callback registrado no then produz outra promise passada adiante
 - A callback retornar uma promise configurada por ela ou um valor qualquer

```
//02c_promiseChain.js
const promise = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Pedido atendido"), 2000);
});

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

console.log("fim do programa")
```

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 exceção)
 - Só pode ser utilizada dentro de funções marcadas com **async**

Async/Await

- Exemplo

```
//03a_simpleAsync.js
async function fazAlgo() {
    let promise = new Promise((resolve, reject) => {
        setTimeout(() => resolve("Pedido atendido"), 2000);
    });
    let resultado = await promise;
    return resultado;
}
```

```
//03a_simpleAsync.js
// ... continuação
console.log("Iniciando o programa")
fazAlgo().then((msg) => console.log(msg))
console.log("Finalizando o programa")
```

```
//03b_simpleAsync.js
// ... continuação
console.log("Iniciando o programa")
fazAlgo().then((msg) => console.log(msg))
console.log("Finalizando o programa")
```

```
//03c_simpleAsync.js
// ... Continuação

async function main() {
    console.log("Iniciando o programa")
    console.log( await(fazAlgo()) )
    console.log("Finalizando o programa")
}

main()
```

- Introdução
- Funções
- Trabalhando com classes
- Exceções
- Funções assíncronas

Considerações finais

Considerações finais

- Explore as referências da disciplina
 - GRONER, Loiane. Learning JavaScript Data Structures and Algorithms. Third Edition. Birmingham: Packt, 2018.
 - HAVERBAKE, Marjin. Eloquent JavaScript. Third Edition. No Starch Press, 2019 .
 - FLANANGAN, David. Java Script: the definitive Guide. Seventh Edition. O'Reilly, 2020.
 - <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript>
- Crie seu repositório/portifólio

E ao final...

SUCCESS

⇒ go get it ⇒

PUCRS online  **uol**edtech.