

Javascript... Do console
a nuvem!



BETHA

O que podemos
esperar do curso de
Javascript?



O que esperar do curso?

- ❑ Diferenciar os ambientes de execução (node X navegador)
- ❑ Sintaxe da linguagem Javascript
- ❑ Ferramentas e bibliotecas
 - ❑ Ferramentas de apoio, build e além
 - ❑ Bibliotecas server-side (back end)
 - ❑ Bibliotecas client-side (front end)
- ❑ Usos do Javascript no console e aplicações servidoras
- ❑ Usos do Javascript no navegador para aplicações web
- ❑ Reconhecer as especificações do Javascript



Configuração do Ambiente

Instalando as ferramentas e configurando o ambiente para trabalhar com Node JS

Vamos conhecer
algumas ferramentas
que facilitarão nosso
trabalho...



Node JS (<https://nodejs.org/>)

Plataforma de execução de códigos Javascript FORA do navegador. Não é apenas um programa, mas um conjunto de programas e bibliotecas que nos permitem realizar uma aplicação completa no desktop/server utilizando como base a linguagem Javascript.

- Utilização da ferramenta **node** para rodar os scripts
- Instala o utilitário npm para:
 - inicializar um projeto Javascript
 - importar biblioteca para o projeto
 - importar utilitários e bibliotecas do node js
- podemos ver os módulos no site: <https://www.npmjs.com/>

Visual Studio Code (<https://code.visualstudio.com/>)

Editor de código fonte leve com diversos plugins (facilitadores) para desenvolvimento de aplicações robustas web.

- Dentre os facilitadores, existe um interpretador de linguagem emmet (<http://emmet.io/>)
- Temos como *debugar* nosso código Javascript que roda sobre o Node JS
- Temos como formatar os arquivos javascript, html e css
- Tem integração com git
- Apesar de desenvolvido pela Microsoft é **Free and Open Source**
- Desenvolvido em [electron](#) e baseado em [atom](#) (editor de código do github)

cmdr (<http://cmdr.net/>)

Emulador de console “um pouco” mais poderoso que o convencional cmd do windows.

- Permite buscar comandos executados no histórico
- Integração com a ferramenta git
- Permite abrir várias abas ao mesmo tempo e com tamanhos de telas diferentes
- Permite aumentar ou diminuir a fonte do console de forma simples (CTRL+Scroll)
- Utiliza como base o ConEmu (outro emulador de console muito famoso)

Tempo para
configurarmos o nosso
ambiente...



Vamos começar a brincar...

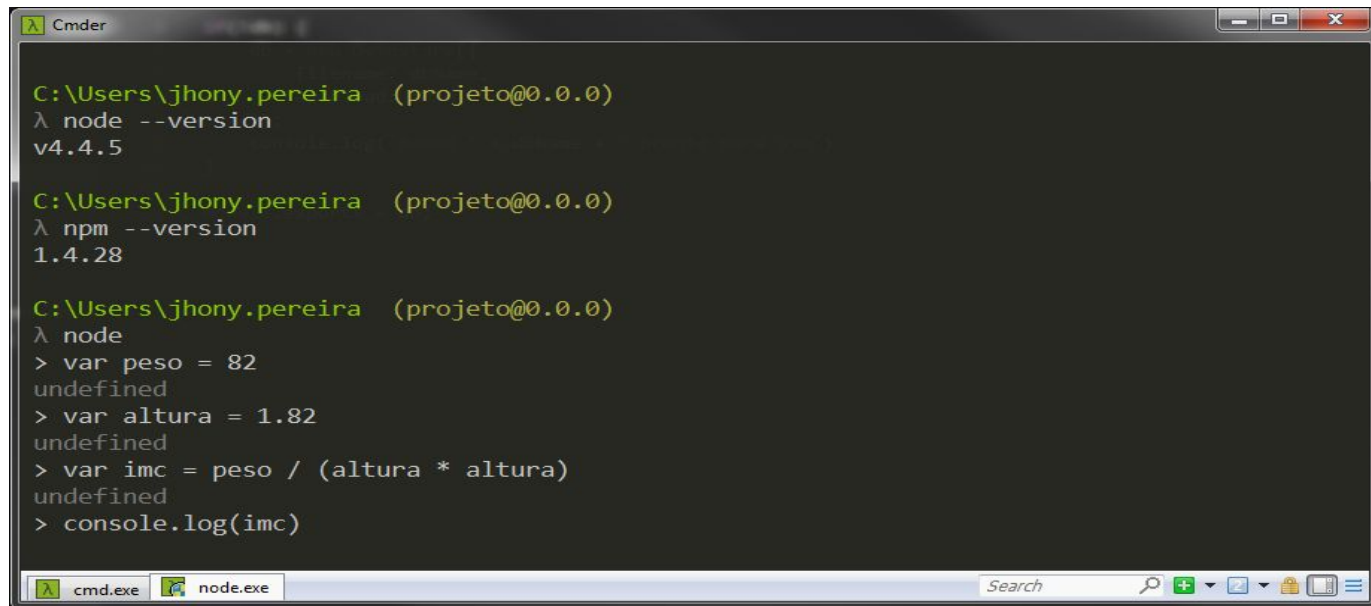




Atividade:

Vamos abrir o **cmd** e verificar se o node está instalado corretamente e aproveite para executar alguns comandos em Javascript.

Repita os passos da imagem abaixo:



```
C:\Users\jhony.pereira (projeto@0.0.0)
λ node --version
v4.4.5

C:\Users\jhony.pereira (projeto@0.0.0)
λ npm --version
1.4.28

C:\Users\jhony.pereira (projeto@0.0.0)
λ node
> var peso = 82
undefined
> var altura = 1.82
undefined
> var imc = peso / (altura * altura)
undefined
> console.log(imc)
```

Sintaxe da Linguagem Javascript

Entendendo a forma e os conceitos que envolvem a
linguagem Javascript

Senta que lá vem história...

A linguagem Javascript surgiu em 1995 através do projeto Netscape Navigator que implementava uma linguagem de script para rodar nas páginas html. O projeto inicialmente tinha o nome de Mocha, depois LiveScript, por fim, acabaram escolhendo JavaScript pela grande popularidade que Java tinha na época.

Mas Java não é JavaScript



Vamos então
desvendar o mistério...



O que é o Javascript?

Javascript é a linguagem padrão dos navegadores web homologada pelo instituto ECMA com o nome de ECMAScript. Está em sua versão 2016*, mas a grande maioria dos navegadores ainda tem problema de suportar esta versão, sendo que a uma versão anterior (ECMAScript 5) é ainda a mais adotada.

Sua sintaxe se parece com Java, mas as comparações acabam por aí, sendo que a linguagem teve mais influências do LISP (linguagem funcional muito famosa nas universidades americanas na década de 80 e 90), pois seu criador era fascinado pela linguagem Scheme (baseada no LISP). Mas teve outras influências como Perl e Self.

* A partir de 2015 o TC39 mudou os nomes das releases de ECMA-262 para serem anuais e suas versões teriam o ano no lugar da edição da linguagem.

Java

- Sintaxe
- Algumas convenções
- Date
- Math

Scheme

- Lambda
- Closure
- Tipagem fraca

Self

- Herança baseada em protótipos
- Objetos dinâmicos

Perl

- Expressões Regulares

Print retirado da apresentação do Rodrigo Branas no seu curso de JavaScript. Você pode conferir todo o curso do Branas neste [link](#).

Quais as características do Javascript?

- Interpretada (não compilada/sem binário ou bytecode)*
- Orientação a objetos baseada em protótipos
- Objetos são dinâmicos
- As funções são de primeira classe
- Tipagem fraca e dinâmica
- Sem suporte a multi-thread
- Tipos simples (primitivos) são tratados como objetos
- É case-sensitive, ou seja, todo o código diferencia letras maiúsculas de minúsculas.



Tipos e declaração de variáveis



Declaração de variáveis

As variáveis em Javascript são declaradas sem definir o seu tipo. O que não quer dizer que ela não é tipada, porém, o que define o tipo de uma variável é o seu conteúdo e não sua declaração.

Algumas regras devem ser seguidas quando declaramos variáveis em Javascript:

- É case-sensitive (diferencia maiúscula de minúscula)
- Deve começar com uma letra, _(underline) ou \$ (cifrão)
- A partir da segunda letra pode ter número, letra, _ ou \$
- É uma boa prática iniciar com letra minúscula e usar o **camelCase**

Quais as tipos de dados em Javascript?

- Tipos primitivos
 - number (todos os numéricos)
 - string (texto/sequência de caracteres)
 - boolean (booleanos ou verdadeiro e falso)
 - null e undefined (tipos especiais)
- Tipos objetos
 - object (objetos de uso geral)
 - array (vetores)
 - function (funções)
 - Date (data)
 - RegExp (expressões regulares*)
 - Error (tratamento de exceções)

*Experimente utilizar o operador **typeof** para verificar os tipos de dados de algumas variáveis ou constantes no console. Ex.: **typeof 13** ou **typeof 'olá'***

Expressões regulares são uma espécie de coringas que identificam elementos em um conjunto de caracteres. Com esse tipo de objeto podemos, por exemplo, validar o formato de um campo telefone ou email.

Tempo para
declararmos algumas
variáveis...



Exercite um pouco realizando as declarações abaixo

```
λ node
> var nome = "Seu Nome"
undefined
> var qtdAmigosFacebook = 5231
undefined
> var _salario = undefined
undefined
> console.log(nome, qtdAmigosFacebook, _salario)
Seu Nome 5231 undefined
undefined
> console.log(typeof nome, typeof qtdAmigosFacebook, typeof _salario)
string number undefined
undefined
> var funcionario = {
... nome: 'João da Silva',
... salario: 2500.52,
... funcao: 'Estagiário'
... }
undefined
> console.log(funcionario)
{ nome: 'João da Silva', salario: 2500.52, funcao: 'Estagiário' }
undefined
> console.log(typeof funcionario)
object
undefined
> |
```

Operadores Aritméticos, Relacionais e Lógicos



Operadores Aritméticos

São utilizados para realizar cálculos matemáticos com números.

A propósito, os números em Javascript seguem uma especificação da IEEE, a IEEE-754 que determina algumas regrinhas de tratamento inclusive de exceções.

var a = 10 + 5; // = 15	var b = 20 - 10; // = 10	var c = 12 / 4; // = 3	var d = 3 * 5; // = 15
Exceções	var e = 0.7 + 0.2; <i>// 0.8999999999999999</i>	var f = 5 / 0; <i>Infinity</i>	var g = 5 / "oi"; <i>NaN</i>

Ainda temos:

Operador de módulo

var modulo = 15 % 4; // = 3

Operadores concatenados

var numero = 2;

numero += 5; numero -= 2; numero /= 2; numero *= 3; numero %= 7;

// numero = numero +5; numero = numero -2; ...

numero++; numero--; *// incremento e decremento*

Operadores Relacionais e Lógicos

São utilizados para realizar comparações e combinações (respectivamente) com entre valores. Para entender como funcionam os operadores lógicos, devemos primeiros entender como funciona os valores `true` e `false` dentro do Javascript.

Para o Javascript temos apenas 6 valores que representam o booleano falso

- **string vazia** (`''`, `""`)
- **número zero** (`0`)
- **valor nulo** (`null`)
- **valores indefinidos** (`undefined`)
- **NaN** (erro de operação aritmética)
- **valor booleano falso** (`false`)

Operadores Relacionais

- `>` e `>=`
- `<` e `<=`
- `==` e `!=`
- `===` e `!==`

Operadores Lógicos

- `&&` (retorna o primeiro *false*)
- `||` (retorna o primeiro *true*)

Exemplos:

```
var texto = "Teste de texto";  
var vazio = "";  
var semValor = null;  
var data = new Date();
```

Tabela verdade JS

- `texto && vazio` = ?
- `data || texto` = ?
- `semValor || data` = ?
- `data && texto` = ?
- `semValor || vazio` = ?
- `data.val || data` = ?

Cuidado com operações booleanas

Existem algumas peculiaridades no comportamento de valores booleanos no Javascript, como a tipagem dinâmica que não funciona com valores string para verificação de operação booleana, conforme visto abaixo:

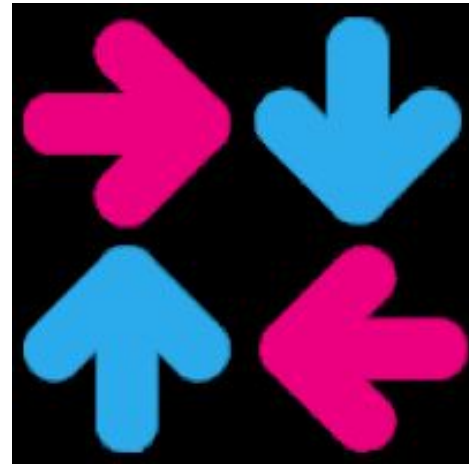
- "0" (zero como string)
- "false" (false como string)
- function() {} (funções vazias)
- [] (arrays vazios)
- {} (objetos vazios)

```
1  false == 0 // -> true
2  false == "" // -> true
3  0      == "" // -> true
```

Verifique também como é o comportamento comparativo entre os diversos tipos de falso no Javascript.

```
1  null == false // -> false
2  null == null  // -> true
3  undefined == false // -> false
4  undefined == undefined // -> true
5  undefined == null  // -> true
```

Estruturas de fluxo/control e repetição



Estruturas de controle

São utilizados para realizar desvios condicionais no código por conta de uma expressão booleana (*if... else*), ou uma lista de opções de uma mesma variável (*switch... case*)

Exemplos de switch... case

Exemplo de if... else

```
if (imc < 18) {  
    msg = "você esta abaixo do peso";  
} else if (imc < 24) {  
    msg = "você esta dentro do esperado";  
} else {  
    msg = "você esta com sobrepeso";  
}
```

```
var juro = 0;  
  
switch (meses) {  
    case 5:  
        juro++;  
    case 4:  
        juro++;  
    case 3:  
        juro++;  
    case 2:  
        juro++;  
    case 1:  
        juro++;  
}
```

```
switch (vogal) {  
    case "a":  
    case "A":  
        console.log("digitou a vogal A");  
        break;  
    case "e":  
    case "E":  
        console.log("digitou a vogal E");  
        break;  
    // outros casos ignorados  
    default:  
        console.log("você não digitou uma vogal");  
}
```

Estruturas de repetição

São utilizados para realizar repetições condicionais ou sequenciais no código por conta de uma expressão booleana (*while/do...while*), uma sequência de iterações ou ainda uma lista de itens (*for*)

Exemplos de while/do...while

```
while (db.hasNext()) {  
    var record = db.next();  
    processaRegistro(record);  
}
```

```
var signal;  
do {  
    signal = realizarLeitura();  
} while(signal === 'on');
```

Exemplos de for/for..in

```
var acumulador = 0;  
for (var i = 0; i < 100; i++) {  
    acumulador += i;  
    console.log("iteração " + i);  
    console.log("acumulador " + acumulador);  
}
```

```
var vetor = ['um', 'dois', 'tres'];  
for (var item in vetor) {  
    console.log("iteração " + item);  
    console.log("valor " + vetor[item]);  
}
```

```
var obj = {nome: 'Teste', idade: 30, valor: 43.53};  
for (var prop in obj) {  
    console.log("prop: " + prop + ", valor: " + obj[prop]);  
}
```

Trabalhando com objetos e funções



Entendendo funções no Javascript

As funções do Javascript são de primeiro nível, isso quer dizer que elas podem ser armazenadas em alguma referência para posteriormente serem chamadas/invocadas. Isso traz um pouco de confusão no uso de funções, pois suas declarações podem ser armazenadas em variáveis, servem de parâmetros para outras funções ou ainda armazenadas em atributos de objetos (métodos).

Exemplos de declaração de funções

```
function processaRegistro(rs) {  
  var obj = new Object();  
  obj.id = rs.get('id');  
  obj.nome = rs.get('nome');  
  return obj;  
}
```

```
var obj = {};  
obj.metodo = function() {  
  console.log("exemplo de método");  
}
```

```
controller.processaRegistro(function(rs) {  
  return {  
    id: rs.get('id'),  
    nome: rs.get('nome')  
  }  
});
```

```
var processaRegistro = function(rs) {  
  var obj = {  
    id: rs.get('id'),  
    nome: rs.get('nome')  
  };  
  return obj;  
}
```


Declaração X Referência

Existe uma diferença de escrever uma função por declaração (function declaration) ou por associação a uma referência (function association/reference).

Na declaração de uma função, seu escopo é do bloco e o processamento acontece antes de ser executado o código, já na referência, a função só existe a partir do momento que a associação da função acontece na execução do código.

Na prática...

```
console.log(primeiraFuncao);  
console.log(segundaFuncao);  
  
function primeiraFuncao() {  
    console.log('oi');  
}  
  
var segundaFuncao = function() {  
    console.log('tchau');  
};  
  
console.log(primeiraFuncao);  
console.log(segundaFuncao);
```

Outras características de funções

- Não pode haver duas funções com o mesmo nome
- Argumentos/parâmetros passados e declarados podem ser diferentes
- Tem escopo próprio e global
- Pode ser criado um novo escopo para ser passado a função
- Contém um vetor com todos os valores passados por parâmetro (**arguments**)
- Argumentos/parâmetros não passados tem o seu valor definido como **undefined**

```
if (!global.minhaFuncao) {  
    global.minhaFuncao = function() {  
        var primeiro = arguments[0];  
        var segundo = arguments[1];  
        var terceiro = arguments[2];  
  
        var media = 0;  
        media += primeiro;  
        media += segundo;  
        media += terceiro;  
        media /= 3;  
  
        console.log('dentro da função', media);  
        return media;  
    };  
}  
  
var media = minhaFuncao(32, 54, 76);
```

Tempo para
declararmos algumas
funções...



Exercite um pouco realizando as funções abaixo

- Função que calcula a média, indiferente da quantidade de parâmetros.
- Função que calcule o IMC de uma pessoa (objeto com peso e altura) ou dos parâmetros peso e altura (numéricos nesta ordem) passados diretamente na função.
- Função que calcule o desconto (segundo argumento) sobre um valor passado (primeiro argumento). Caso não seja passado um desconto, o desconto padrão é 25%.
- Função que calcule o desconto do IRRF (baseado na tabela atual) de um funcionário passado como parâmetro e crie o campo no objeto funcionário para armazenar o valor.
- Criar uma função que execute operações (função passada como primeiro argumento) com outros dois argumentos passados.

Objetos em Javascript

A Orientação do Javascript é dinâmica e baseada em protótipos, isto permite que os componentes do objetos sejam construídos na medida que vamos precisando deles. para adicionar um novo método ou atributo de um objeto, basta atribuí-lo e ele já estará pronto para uso. Abaixo alguns exemplos de declaração de objetos em Javascript.

```
var obj = {};  
obj.nome = 'Nome';  
obj.idade = 35;
```

```
var obj = new Object();  
obj.nome = 'Nome';  
obj.idade = 43;
```

```
var Obj = {  
  nome: 'Nome',  
  idade: 32  
};
```

Exemplos de declaração de objetos

```
// Exemplo de função construtora  
function MeuObjeto(nome, idade) {  
  this.nome = nome;  
  this.idade = idade;  
}  
var obj = new MeuObjeto('Nome', 25);
```

```
// Exemplo de função produtora  
function criaMeuObjeto(nome, idade) {  
  return {  
    nome: nome,  
    idade: idade  
  };  
}  
var obj = criaMeuObjeto('Nome', 33);
```

Herança em Javascript

Como Javascript é baseado em protótipos, quer dizer que ele não tem classe e a reutilização de comportamento (herança) ocorre do processo de expandir objetos existentes que servem como modelos. Abaixo um exemplo de herança em Javascript.

```
function Pessoa(nome) {  
    this.nome = nome;  
}  
  
Pessoa.prototype.dizOi = function() {  
    console.log('Olá, meu nome é', this.nome);  
}  
  
Pessoa.prototype.dizTchau = function() {  
    console.log('Tchau!!!');  
}
```

```
function Funcionario(nome, cargo) {  
    Pessoa.call(this, nome);  
    this.cargo = cargo;  
}  
  
Funcionario.prototype = new Pessoa;  
Funcionario.prototype.constructor = Funcionario;  
  
Funcionario.prototype.dizOi = function() {  
    console.log('Olá, me chamo', this.nome,  
                'e trabalho no cargo', this.cargo);  
}
```

Tempo para
declararmos alguns
objetos...



Exercite um pouco realizando a atividade abaixo

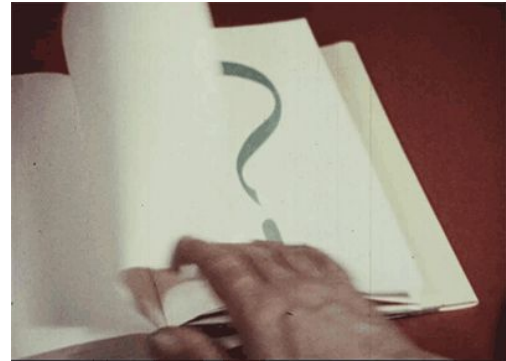
- Implemente os modelos do slide anterior e teste o comportamento de instâncias de cada um dos objetos baseados neste modelo conforme a figura abaixo:

```
var pedro = new Pessoa('Pedro');
pedro.dizOi();
pedro.dizTchau();

var joao = new Funcionario('João', 'Estagiário');
joao.dizOi();
joao.dizTchau();

console.log('João é instancia de Pessoa', joao instanceof Pessoa);
console.log('João é instancia de Funcionario', joao instanceof Funcionario);
console.log('Pedro é instancia de Pessoa', pedro instanceof Pessoa);
console.log('Pedro é instancia de Funcionario', pedro instanceof Funcionario);
```


Vamos responder alguns questionamentos...



Vamos relembrar alguns itens respondendo a alguns questionamentos...

- Qual o real nome de Javascript e porque tem dois nomes?
- Quais os tipos primitivos do Javascript e como eles se comportam?
- Quais os erros comuns com operações matemáticas em Javascript?
- Quais os valores true e false para o Javascript?
- Como podemos declarar uma função em Javascript?
- Como funciona a declaração de um objeto em Javascript?
- Como podemos estender funcionalidades de um objeto pai para um objeto filho em Javascript?
- O que é a propriedade prototype das funções em Javascript?

Trabalhando com módulos em Javascript

Organize seu código fonte de forma efetiva e isolada
para uma programação eficiente

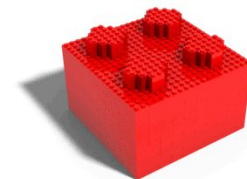
O que são módulos?



O que é um Módulo em Javascript?



- Uma forma de organizar seu código fonte
- Uma forma de utilizar bibliotecas de terceiros e da própria plataforma (core modules e wrapper functions)
- Permite encapsular uma funcionalidade, ou até mesmo uma biblioteca inteira (reaproveitamento e estabilidade do código)
- Permite subdividir o código fonte para uma melhor manutenibilidade e isolamento de escopos (granularidade e namespacing)
- Várias soluções criadas para o mesmo propósito (ex.: CommonsJS, AMD)
- Até a versão ES5 não existia uma forma padrão da linguagem
- Na versão ES6 (ES2015) a linguagem veio com este recurso



Padrões de Módulos: CommonJS

- Utiliza importação síncrona de módulos
- Os módulos são de primeira classe e armazenados em referências
- É o padrão utilizado pelo Node JS
- Cada arquivo no projeto é um módulo e tem em seu escopo um objeto **module.exports** (ou **exports**) para expor o que é público
- Utiliza a função global **require** para importar um módulo

```
// financeiro.js
function desconto(valor, desconto) {
    return valor * ( 1 - desconto / 100 );
}

function acrescimo(valor, acrescimo) {
    return valor * ( 1 + acrescimo / 100 );
}

module.exports = {
    desconto: desconto,
    acrescimo: acrescimo
}
```

```
// main.js
var f = require('./financeiro');

var valor = f.acrescimo(250, 15);
console.log(valor);
```

Padrões de Módulos: AMD

- Asynchronous Module Definition (definição assíncrona de módulos)
- Módulos são injetados nos seus dependentes
- É o padrão utilizado pela biblioteca require.js (base para o angular)
- Usa a função define para especificar e injetar as devidas dependências
- É browser-first, o que não permite seu uso em ambientes como Node JS

```
// financeiro.js
define('financeiro', [], function(){
    return {
        desconto: function() {
            return valor * ( 1 - desconto / 100 );
        },
        acrescimo: function() {
            return valor * ( 1 + desconto / 100 );
        }
    }
});
```

```
// main.js
define(['financeiro'], function(financeiro) {
    var valor = financeiro.desconto(250, 15);
    console.log(valor);
});
```

Padrões de Módulos: Native JS / ES Harmony

- Especificação de módulos nativo em Javascript
- Definido pelo TC39 na especificação de ECMAScript 6
- Seguem um padrão muito próximo do CommonJS
- Permitem importação por referência
- Usa a palavra reservada **export** para expor seus contextos e **import** para importar um contexto de um módulo

```
// financeiro.js
export function desconto() {
    return valor * (1 - desconto / 100);
}
export function acrescimo() {
    return valor * (1 + desconto / 100);
}
```

```
// main.js
import * as financeiro from './financeiro';

var valor = financeiro.desconto(250, 15);
console.log(valor);
```


Tempo para
declararmos alguns
módulos...



Exercite um pouco realizando a atividade abaixo

- Implemente o módulo financeiro e crie um código que use este módulo.
- Implemente outros métodos que você considera interessante no módulo financeiro e utilize seus novos métodos.



Node JS: core modules

- Módulos embarcados no Node JS
- Expressam funcionalidades e bibliotecas da plataforma
- Podem ser importados no padrão CommonJS ou Native JS
- Usa-se na sua importação apenas o nome do módulo, ignorando a sua localização (preocupação do Node JS)

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

var pastaBase = 'c:/windows';
var regex = /\.log$/i

fs.readdir(pastaBase, function (erro, arquivos) {
  if (erro) {
    console.log("Não foi possível listar %s", pastaBase);
    return;
  }
  arquivos.forEach(function (nomeArquivo) {
    var file = path.join(pastaBase, nomeArquivo);
    if (regex.test(nomeArquivo))
      console.log("%s (%s)", nomeArquivo, file);
  });
});
```

Node JS: bibliotecas de terceiros

- Complementar aos core modules
- Funcionalidades ou ferramentas para utilização nas aplicações
- Tem sua importação na aplicação igual aos core modules (apenas o identificador)
- Sua instalação deve ser feita por um gerenciador de dependências (por exemplo o npm)

```
λ npm install express
C:\curso\javascript\material\projeto
-- express@4.15.1
+-- accepts@1.3.3
| +-- mime-types@2.1.14
| | -- mime-db@1.26.0
var http = require('http');

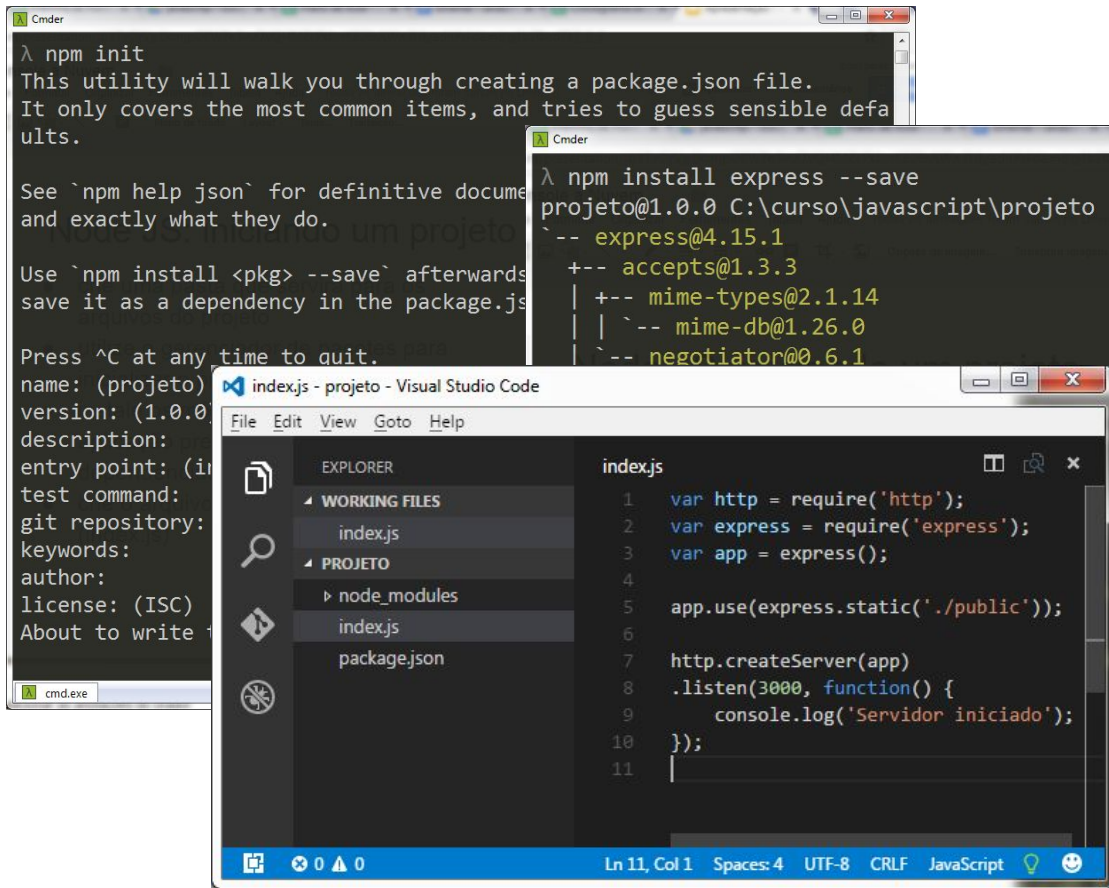
var express = require('express');
var app = express();

app.use(express.static('./public'));

http.createServer(app)
  .listen(3000, function() {
    console.log('Servidor iniciado');
  });
```

Node JS: iniciando um projeto

- crie uma pasta que servirá para os arquivos do projeto
- utilize o gerenciador de pacotes para inicializar o projeto (***npm init***)
- instale os módulos adicionais que sua aplicação precisa, salvando as dependências no projeto (***npm i --save***)
- crie o arquivo principal do projeto (***index.js***)



The screenshot displays the initial steps of setting up a Node.js project. On the left, a Command Prompt window shows the execution of `npm init`, which prompts for project details like name, version, and description. Below this, the `npm install express --save` command is run, installing Express and its dependencies (accepts, mime-types, mime-db, negotiator) into the `node_modules` directory. On the right, the Visual Studio Code editor is open, showing the `index.js` file in the `PROJETO` folder. The file contains a simple Express server setup that listens on port 3000 and logs 'Servidor iniciado'.

```
λ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (projeto)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to package.json
```

```
λ npm install express --save
projeto@1.0.0 C:\curso\javascript\projeto
-- express@4.15.1
+-- accepts@1.3.3
| +-- mime-types@2.1.14
| | -- mime-db@1.26.0
| -- negotiator@0.6.1
```

```
index.js - projeto - Visual Studio Code
File Edit View Goto Help

EXPLORER
WORKING FILES
  index.js
PROJETO
  node_modules
  index.js
  package.json

index.js
1 var http = require('http');
2 var express = require('express');
3 var app = express();
4
5 app.use(express.static('./public'));
6
7 http.createServer(app)
8   .listen(3000, function() {
9     console.log('Servidor iniciado');
10  });
11
```

Tempo para criarmos
o nosso projeto...



Padrões de Implementação em Javascript

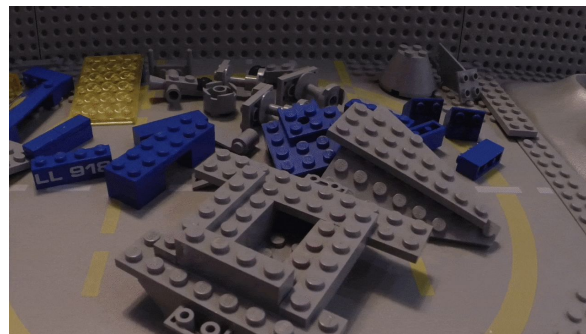
Melhore a eficiência e reusabilidade de seu código
usando modelos conceituados

O que são padrões de implementação?



O que são padrões de implementação?

- São soluções reutilizáveis para problemas recorrentes
- Já explorados para diversos problemas (maturidade da solução)
- Permitem o fácil entendimento através de um vocabulário padrão e métodos padronizados de implementação
- Podem ser escritos em qualquer linguagem ou plataforma
- Previnem possíveis problemas que podem ocorrer no desenvolvimento de uma solução ad-hoc (específica para uma finalidade)
- Facilitam a legibilidade de código
- Tem vasta documentação
- Exemplos diversificados



Quais os padrões de implementação em Javascript?

Por ser uma plataforma que preza pela realização assíncrona de tarefas, o Javascript tem certos padrões de implementação que merecem nossa atenção.

Além dos já conhecidos design patterns (GoF) que podem ser implementados também em Javascript, temos alguns padrões que devemos ir um pouco mais a fundo, pois são muito utilizados as soluções desta plataforma.

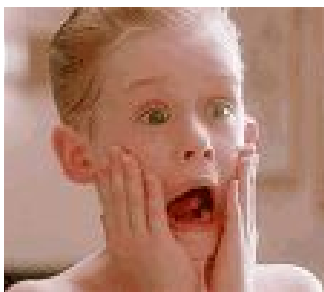




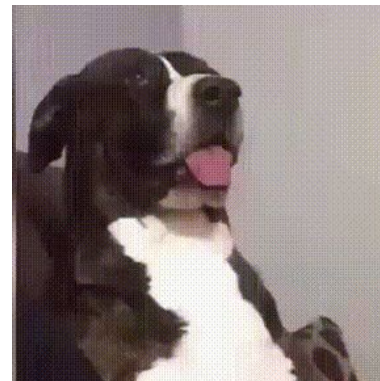
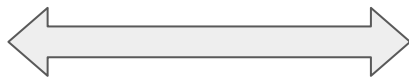
Programação Orientada a Eventos

O que são eventos em Javascript?

- Os eventos são uma das razões de ser o Javascript.
- Eles permitem vincular código a determinadas ações que ocorrem nos objetos.
- Clicar sobre um botão ou realizar a conexão com um banco de dados são exemplos de eventos que podemos “estar escutando”.



Subject



Observer

Disparando ações a partir de eventos

Para se utilizar um evento, deve-se observar a documentação do objeto ou API que se está utilizando para entender como a função é “ligada” ao evento.

Muitas implementações têm métodos padrões para cada um dos eventos que devem ser associados, outras utilizam o método “**on**” com uma string indicando o evento ao qual está ligada a função.

```
btn.onclick = function() {  
    alert('Você clicou no botão');  
};
```

```
connection.on('error', function (error) {  
    console.log('Erro na conexão: ' + error);  
});
```

```
input.onblur = function() {  
    alert('Você deixou o campo');  
};
```

```
db.on('disconnected', function() {  
    console.log('O banco de dados foi desconectado');  
});
```



Tratando ações assíncronas com callbacks



Ações assíncronas com callbacks

Damos o nome de callback as funções que são passadas como parâmetro para executar determinado tratamento dentro de uma outra função.

Isso só é possível no Javascript por conta de sua característica de trabalhar com funções de primeiro nível.

A função passada como parâmetro (declaração) é acionada dentro da função principal (invocação).

```
function consultaBanco(sql, trataRegistro) {  
  db.query(sql, function(err, data) {  
    if (err) {  
      trataRegistro(err);  
    } else {  
      trataRegistro(null, data);  
    }  
  });  
}
```

Invocação

```
var lista = [];  
consultaBanco("SELECT * FROM CIDADES", function(err, data) {  
  if (!err) {  
    lista.push({  
      nome: data.get("nome"),  
      estado: data.get("estado")  
    });  
  }  
});
```

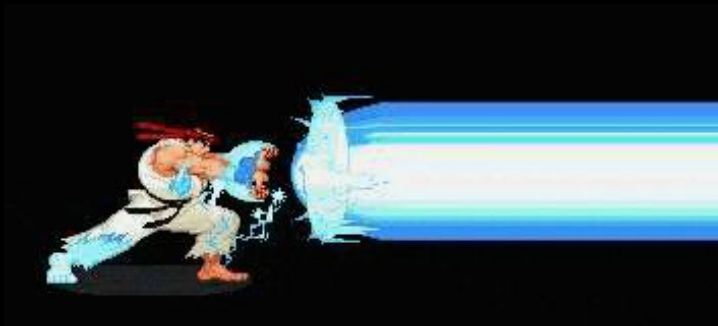
Declaração

Mas qual o problema
com os callbacks?



The Callback Hell, ou “código *hadouken*”

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





Organizando ações assíncronas com Promises

O que é uma Promise?

Uma Promise é um objeto que encapsula uma execução assíncrona com objetivo de melhorar a organização do código fonte.

Esta implementação permite encapsular os retornos em funções distintas e não em posição de parâmetros, como é feito nos callbacks.

Uma Promise, é uma implementação robusta com vários facilitadores e um deles é o estado da promise que é representado por três situações: pendente, realizada e rejeitada.



Como utilizamos uma Promise

O objeto Promise tem um método **then** que associamos a função que trata o estado de realizada (quando a Promise executa sem erros) e nesta função é passado o resultado da operação.

No método **catch** do objeto Promise é passada a função que trata o retorno de erros na execução.

Também podemos tratar os erros em cada um dos passos da Promise (**then**) para isso, basta passar como segundo argumento a função de tratamento de erro.

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

module.exports = function(file) {
  return new Promise( function (resolve, reject) {
    fs.readFile(file, 'utf-8', function(err, res) {
      if (err) {
        reject(err);
      } else {
        resolve(res);
      }
    });
  });
};

var readFile = require('./read-file');
```

```
readFile('./file.json')
  .then(JSON.parse)
  .then(function(obj) {
    console.log('resultado:', obj)
  })
  .catch(function(err) {
    console.log('erro:', err)
  });
```

ECMAScript 6, ES 6 ou ES2015

A mudança mais significativa na linguagem
Javascript