E como podemos fazer o mesmo entendimento em JavaScript? Ou seja, como a linguagem funciona?

A linguagem JavaScript surgiu mais ou menos ao mesmo tempo que o Java (1995) e começou a ser adotada mais ou menos ao mesmo tempo (1997).

O nome original da linguagem foi LiveScript, e foi criada para ser uma linguagem que traria vida ao HTML, ou seja, permitiria ao usuário de um browser interagir dinamicamente com uma página estática.

Foi criada por Brendam Eich, na época trabalhando na Netscape, a quem normalmente atribuímos o crédito da criação da linguagem. Seguindo a esteira de sucesso do Java, a linguagem foi renomeada para JavaScript, o que fazia sentido, uma vez que ambas tinham uma mesma linguagem de origem, ou seja, podemos visualizar JavaScript como uma linguagem Java "relaxada" em termos de formalidade.

Embora as duas linguagens tenham nomes similares e sigam uma mesma sintaxe da linguagem C++, os princípios que nortearam sua criação foram distintos e, portanto, há grandes divergências funcionais. Java é uma linguagem fortemente tipada. Ou seja, se você declarar uma variável, ela deverá armazenar este mesmo tipo até ser destruída. Não há como mudar o tipo de uma variável em Java após criada. Em JavaScript, as variáveis são fracamente tipadas, ou seja, você pode atribuir o que quiser na variável, e a linguagem não irá se queixar. Na verdade, você nem precisa (nem deve) declarar o tipo dela. Esta característica seria mais adequada a uma linguagem interpretada em execução em um browser.

Em Java as variáveis estão "presas" dentro de Objetos e

definidas dentro de classes, enquanto que em JavaScript você tem mais liberdade para declaração de variáveis.

Outra grande diferença é que Java é orientada para objetos (OOP). Não é considerada totalmente OOP, apenas porque os tipos básicos da linguagem (inteiros, flutuantes, etc.) não são objetos. Uma linguagem como SmalTalk já era considerada totalmente OOP em 1960, ou seja, estes conceitos de OOP são muito sólidos.

No caso do JavaScript, hoje se utiliza o termo orientada a "prototypes", ou baseada em protótipos, ao invés de "object oriented" como o Java. Discutiremos mais sobre estas implicações em outros capítulos.

No caso do Hello World, em JavaScript (no browser) seria algo como:

```
<!DOCTYPE HTML>
<html>
    <body>
        <script>
            alert('Hello, World!');
        </script>
    </body>
</html>
```

Não precisamos gastar muito tempo entendendo este dialeto, pois o JavaScript mais importante é aquele que executa no servidor, mas alguns detalhes importantes podem ser vistos. Por exemplo, se você conhece algo de HTML, pode perceber que este exemplo é uma página de browser. Uma técnica comum, para chavear entre conteúdo estático e conteúdo dinâmico é adicionar uma TAG que faça este chaveamento.

Esta é a função da TAG **<script>**.

Mas as boas práticas sugerem que se deixe todo código JavaScript em um arquivo separado

```
<script src="myScript.js"></script>
```

Neste caso todos os códigos ficam em um arquivo texto externo, que pode ser carregado e interpretado sob demanda. Ou seja, JavaScript não passa por um processo formal de compilação, como o Java. A linguagem é interpretada (no sentido formal da palavra), mas os defensores amenizam dizendo que ela é compilada através de um JIT, como o existente no Java. A diferença é que no Java o JIT trabalha com os bytecodes previamente compilados e validados, enquanto que no JavaScript o JIT compila e otimiza diretamente os códigos fontes.

Esta característica torna códigos Java mais "robustos", e aqui precisamos entender este conceito, para que eu não seja massacrado pelo time JavaScript. Muitos problemas do Java já são detectados no momento da compilação. Usos indevidos de variáveis, atribuições errôneas, passagens de parâmetros para métodos inválidas, etc. Tudo isto é validado em tempo de compilação, e você nem conseguirá colocar os códigos para executar.

Já no caso do JavaScript, como os códigos fontes são interpretados durante a execução, qualquer digitação ou distração somente será detectado no momento da execução.

Acredito que o JavaScript exige um pouco mais de cuidado (ou habilidade) durante o processo de criação, o que

pode ser auxiliado por um bom editor de texto que faça partes destas validações.

Uma questão interessante é como o JS do browser consegue interagir com uma página? Bem o HTML é uma especialização de um XML, que é composto por TAGs de abertura e TAGs de fechamento. Veja no exemplo, temos **<html>** e **</html>**. Dentro dele temos um **<body>** e **</body>**. Quando o HTML é carregado pelo browser, ele gera uma árvore com todo o conteúdo, chamada de árvore DOM. O JavaScript tem a capacidade de navegar nesta árvore buscando por elementos específicos e modificando ou interagindo com seu conteúdo, como em:

```
document.getElementById('body').append('<div>Ola
mundo!</div>');
```

Além disto, pode-se receber eventos de interações com o usuário, como interações com o mouse ou digitação de teclas. A junção destes dois recursos permite que se tenha dinamismo nas páginas. Por isto, hoje JavaScript é a linguagem de programação mais utilizada no mundo. Praticamente cada página WEB no planeta terra contém algum tipo de animação ou código JavaScript. Quase sempre que você vê algo se mover na tela ou interagir com o usuário em um browser, é quase certo que exista um código JavaScript.

Outra característica bastante interessante do JavaScript é que como ela foi criada para interagir com o browser, e portanto tende a ser leve e não bloqueante, ou seja, tudo acontece de forma assíncrona. Também por causa desta mesma origem, eventos são utilizados para quase tudo dentro

da linguagem.

Como foi extensamente utilizado, houve um grande período para sua maturação, desde sua criação. O Google investiu esforços extras no seu browser Chrome, e alguns especialistas dizem que o "motor" de JavaScript do Chrome (chamado "V8") é o mais eficiente existente.

Mais recentemente, em 2008, Ryan Dahl teve uma excelente sacada, que foi isolar o motor do V8 e coloca-lo para executar como um programa externo ao browser. Isto permite que se crie programas utilizando a linguagem JavaScript do lado servidor, e ainda com algumas bibliotecas que facilitam o uso na criação de aplicativos corporativos. Este foi o surgimento do Node.JS, de uma forma resumida. Ou, seja, excetuando-se algumas características específicas de implementação, programar em Node.JS é essencialmente programar em JavaScript.

Isto pode ser vantajoso para criação de aplicações Web, porque a linguagem das páginas Web pode ser a mesma utilizada na infraestrutura do Servidor.

Vamos estudar algumas destas características essenciais da linguagem, e posteriormente estudaremos a sintaxe dela.

Como JavaScript não têm o conceito de Objetos, tudo é global. Cada variável declarada fica em um mesmo local. Em um programa complexo, isto é inaceitável, já que a possibilidade de utilizarmos um mesmo nome de variável para algo já existente lá no começo dos códigos é muito grande. Seria como tentar escrever um livro sem repetir uma única palavra. Para resolver este problema e criar algo modular, foi utilizado um recurso interessante da linguagem.

Embora JS não tenha objetos, ele tem métodos, e podemos dizer que são mais poderosos do que os métodos em Java, que ficam presos dentro de classes.

```
function test(first, last, age, eye) {
   this.firstName = first;
   this.lastName = last;
   this.age = age;
   this.eyeColor = eye;
}
```

Normalmente um método é chamado, processa algo e termina sua execução, possivelmente retornando algo a quem o chamou. Mas podemos fazer uso de um recurso interessante, que é chamado em JavaScript de "closure". Dentro de um método, existe um espaço onde variáveis e funções podem existir como um universo a parte, sem interferir no restante do programa. Além disto, podemos criar vários destes pequenos universos e utilizá-los para criar códigos complexos. Por exemplo, com estas linhas:

```
var var1 = new test("John", "Doe", 50, "blue");
var var2 = new test("Sally", "Rally", 48, "green");
```

Acabamos de criar dois destes universos, capturados em var1 e var2. Os atributos **firstName**, **lastName** estão isolados nas duas instâncias, dentro dos métodos, não conflitando entre si. Este recurso é interessante em JS, sendo similar a uma classe Java, e no JavaScript é fundamental para criar códigos de qualidade, e ainda é a base de todo o Node.JS. Para acessar o campo **firstName** dos dois objetos podemos fazer:

```
var name1 = var1.firstName;
var name2 = var2.firstName;
```

Como dissemos podemos colocar métodos dentro de métodos em JavaScript. Algo como:

```
function test(first, last, age, eye) {
   this.firstName = first;
   this.lastName = last;
   this.age = age;
   this.eyeColor = eye;
   function getFullName() {
      return firstName + " " + lastName;
   }
}
```

Que podem ser acessados através de

```
var fullname = var1.getFullName();
```

neste caso retornando **"John Doe"**, ou a concatenação das Strings. Estes universos isolados criados dentro de métodos mantêm alguma similaridade com Java e permitem implementar um tipo de orientação para objetos.

Este recurso poderia até ser chamado de **"orientação para métodos"** (inventei isto agorinha), e nos permite criar unidades autônomas de programação, podendo gerar programas bastante complexos.

Mas como criamos um programa em Node.JS? Vamos fazer um programa bem simples. Se você criar um arquivo test.js com o conteúdo:

```
console.log("Hello World");
```

e chamar a linha de comando **node test.js**, verá o resultado da execução sendo apresentada na tela. Ele não precisa de passos intermediários para compilação como o Java, e basta salvar o texto e chamar a linha de comando.
Ele não é muito útil, portanto vamos criar um módulo que faz algumas operações mais complexas.
Se você mudar o test.js para

```
var mat = require("matematica");
console.log(mat.soma(10,10));
```

O **require** indica que você precisa de um módulo externo ao seu programa, e neste caso ele deve ser atribuído à variável mat. Mas de onde vem este módulo matematica? Iremos cria-lo agora mesmo.
Por convenção os módulos em node.js ficam dentro de um diretório node_modules. Podemos criar este diretório no mesmo nível onde o test.js se encontra, e dentro dele podemos criar um diretório matematica. Dentro deste diretório irá o código do módulo, que pode ficar por exemplo dentro de um arquivo com o nome index.js. A estrutura toda fica, portanto:

```
.
|____node_modules
| |____matematica
| | |____index.js
|____test.js
```

Ou seja, a leitura para esta construção é que temos um programa index.js que tem o módulo matematica sendo utilizado como dependência.
Dentro do arquivo index.js no diretório matemática vai:

```
exports.soma = function(num1, num2) {
  return num1 + num2;
}
exports.subtracao = function(num1,num2) {
  return num1 – num2;
}
```

Quando você coloca um **require('modulo')**, o node irá tentar encontrar algum módulo com este nome dentro de **node_modules**. Se conseguir tenta encontrar algum arquivo que possa ser interpretado por ele. O nome index.js está na lista de busca do node, e ele carrega o módulo.

Mas de onde surgiu o **exports** dentro de index.js? E pior, como podemos garantir que o conflito entre módulos não irá acontecer? Por exemplo, se modificarmos index.js para:

```
exports.soma = function(num1, num2) {
  return num1 + num2;
}
var variavel_global = 0;

exports.subtracao = function(num1,num2) {
  return num1 – num2;
}
```

Como podemos ter certeza de que **variavel_global** não será vista de forma global por todos os outros módulos?
Bem, existe um truque aqui, implementado pelo Node.JS.

Quando o método **require** é chamado, internamente o Node.js encapsula tudo o que estiver dentro do arquivo fonte do módulo (index.js neste caso) em um método com a assinatura:

```
(function (exports, require, module, __filename, __dirname) {
    <conteudo do arquivo js>
})
```

A variável **mat** tem, portanto, o conteúdo:

```
(function (exports, require, module, __filename, __dirname) {
     exports.soma = function(num1, num2) {
          return num1 + num2;
     }
     var variavel_global = 0;
     exports.subtracao = function(num1,num2) {}
})
```

Percebe como o conceito de "closure" está criando um pequeno universo isolado para cada módulo? Ou seja, **variável_global** fica presa dentro do método, e visível por todo mundo dentro dele, mas não conflita com o que estiver fora, como outros métodos. Uma bela decisão de arquitetura de software, creio eu.

Tudo o que for colocado na variável **exports** ou **module.exports** será retornado para quem chamou **require**.

Outras informações disponíveis no método são:

exports – o que será retornado como exportado
require – usado internamente para carregar as dependências do módulo
module – informações sobre o módulo

```
__filename – nome do arquivo js carregado para este módulo
__dirname – diretório de onde o JS foi carregado para este módulo
```

Basicamente existem duas formas de usar este recurso:
Podemos declarar cada método como uma atribuição de **exports**, como fizemos com **exports.soma** e **exports.subtracao**.
A segunda forma é atribuirmos diretamente um método a **exports**, desta forma:

```
//dentro de index.js
exports = function(numero1, numero2) {
   return numero1 + numero2;
}
```

E do lado cliente utilizaríamos como

```
 var mat = require("matematica");
console.log(mat(10,20));
```

Talvez seja algo intuitivo, mas se o módulo matematica precisar de outros módulos, dentro do diretório matemática haverá um diretório **node_modules**, e dentro dele, os módulos necessários para que ele funcione, e assim recursivamente.

Ok, mas como gerenciar vários arquivos JavasScript dentro de um módulo, ou como gerenciar versões do módulo? Dentro de cada diretório de módulo existe um arquivo descritor, com o nome **package.json**. Ele não é mandatório, e nosso exemplo anterior funcionou sem ele, mas será necessário caso precisemos fazer ajustes finos nos módulos. O formato dele é um JSON com várias informações sobre o módulo:

```
{
 "name": "module-name",
 "version": "10.3.1",
 "description": "An example module to illustrate the usage of a package.json",
 "author": "Your Name <you.name@example.org>",
 "contributors": [{
 "name": "Foo Bar",
 "email": "foo.bar@example.com"
}],
 "bin": {
 "module-name": "./bin/module-name"
},
 "scripts": {
  "test": "vows --spec --isolate",
  "start": "node index.js",
  "predeploy": "echo im about to deploy",
  "postdeploy": "echo ive deployed",
  "prepublish": "coffee --bare --compile --output lib/foo src/foo/*.coffee"
 },
 "main": "lib/foo.js",
 "repository": {
 "type": "git",
 "url": "https://<url>"
},
 "bugs": {
 "url": "https://<url>"
},
 "keywords": [
 "keyword1",
 "keyword2"
],
 "dependencies": {
  "async": "~0.8.0",
  "express": "4.2.x",
  "winston": "git://<url>",
  "bigpipe": "bigpipe/pagelet"
},
```

```
 "devDependencies": {
  "vows": "^0.7.0",
  "assume": "<1.0.0 || >=2.3.1 <2.4.5 || >=2.5.2 <3.0.0",
  "pre-commit": "*"
 },
 "preferGlobal": true,
 "private": true,
 "publishConfig": {
 "registry": "https://<url>"
},
 "subdomain": "foobar",
 "analyze": true,
 "license": "MIT"
}
```

Ele contém uma quantidade bem grande de campos, mas pode-se começar bem pequeno, algo como:

```
{
 "name": "nome-do-modulo",
 "version": "1.3.1",
 "description": "Descrição do módulo",
 "main": "index.js"
}
```

O campo **version** é importante porque adiciona a capacidade de versionamento. O campo **main** aponta para o arquivo que irá conter o JS de entrada do nosso módulo (ou seja, não precisamos utilizar index.js). Vários campos como **email**, **contributors**, **author**, por exemplo, são importantes porque existe um repositório global de módulos node.js, e se você deseja ser reconhecido pelo seu trabalho, estas informações ficam por lá quando você publicar seu módulo.

Aqui é interessante comparar com Java. Veja que cada módulo tem suas dependências dentro do diretório node_modules, que fica dentro do próprio módulo, e os problemas de versão de classes com Java serão minimizados em Node.JS.

Uma seção a destacar é **dependencies**. Dentro dela estão os módulos e versões necessárias para o funcionamento deste pedaço de aplicação. É importante que sejam descritos os módulos necessários, porque quando formos distribuir nossa aplicação, não precisamos empacotar tudo o que é necessário para que ela execute. Após instalar o Node na sua máquina, fica disponível um outro comando **npm**, que é o gerenciador de módulos do Node.

Quando você chama o comando **npm init**, dentro do diretório do aplicativo, ele baixa todas as dependências do módulo em cascata, a partir de um repositório central (o mesmo que você pode utilizar para publicar seu módulo).

Entre as várias opções, ele permite baixar e instalar dependências, publicar um novo módulo no diretório global, buscar e listar APIs disponíveis. Por isto é desejável utilizar esta abordagem descentralizada de uso dos módulos.

Na minha máquina, por exemplo, se digito **sudo npm list -g** ele retorna uma lista de módulos que foram instalados globalmente. Os módulos locais são aqueles que ficam instalados dentro do diretório da minha aplicação (matematica por exemplo) e os módulos globais ficam instalados em algum local do sistema operacional, disponível para uso por todos os programas. Quando você executa um programa, ele tenta encontrar localmente o módulo, e caso

não consiga parte para o escopo global.

O Node.js é utilizado como base de outras tecnologias mais recentes, como Cordova e IONIC. O MFP (Mobile First Platform) da IBM, utiliza o node.js para as ferramentas de linha de comando.

Vários serviços do BlueMix fornecem clientes para interagir com os serviços utilizando módulos Node.JS.

Quando você cria um aplicativo IONIC, por exemplo, está interagindo com o Node.JS. As ferramentas linha de comando do IONIC, Cordova e MFP da IBM são módulos Node.JS instalados globalmente que podem ser chamados a partir da linha de comandos.

Ou seja, vale a pena gastar esforços para aprender Node!