GRADUAÇÃO EM ANÁLISE E DESENVOLVIMENTO

PARADIGMAS DE PROGRAMAÇÃO: REACT NATIVE

Aula 01: Conceitos Básicos – JavaScript (ES6)

Obs.: *os códigos apresentados nesse documento foram codificados e testados na ferramenta* http://jsfiddle.net *- os códigos exemplo do documento podem ser baixados* *aqui.*

INTRODUÇÃO

O conteúdo dessa aula tem por objetivo revisar (ou apresentar) conteúdos básicos sobre a linguagem *JavaScript*, abordando os principais conceitos disponibilizados pela ECMAScript 6 – ES6. Esse conhecimento prévio será necessário a partir da aula 03, quando conceitos serão utilizados e aplicações *React Native* serão desenvolvidas. Como essa disciplina é ofertada aos alunos dos últimos anos do curso, considera-se que conceitos prévios de programação, como laço, *array* e objeto, já são conhecidos.

FUNÇÕES JAVASCRIPT – ARRAY

map: A função "map" é utilizada para traduzir/mapear todos os elementos de um *array* para algum outro conjunto de valores. Ela percorre o *array* da esquerda para a direita invocando uma função de retorno (com parâmetros) para cada elemento percorrido. Para cada chamada de retorno, o valor devolvido se torna elemento do novo *array*. Após todos os elementos serem percorridos, o *map()* retorna o novo *array* com todos os elementos "traduzidos".

Protótipo

```
array.map( function( elem, index, array ) {
      ...
}, thisArg );
```

Parâmetro	Significado
elem	Valor do elemento
index	Índice em cada iteração, da esquerda para a direita
array	*Array* original invocando o método
thisArg	(opcional) Objeto que será referenciado como *this* no *callback*

Exemplo de Codificação

```
// Exemplo utilizando Array Simples
var metros = [ 1, 5, 3, 2 ];
```

```
var milimetros = metros.map( function( elem ) {
	return elem * 1000;
});

for(var a=0; a<milimetros.length; a++) {
	document.write(milimetros[a] + ' mm<br>');
}

// Função "milimetros" utilizando Sintaxe ES6
var milimetros = metros.map( elem => elem * 1000 );

// Exemplo utilizando Array de Objetos
var cidades = [
	{ nome : 'Maringá', temperatura : 32 },
	{ nome : 'Curitiba', temperatura : 18 },
	{ nome : 'Paranaguá', temperatura : 35 },
	{ nome : 'Londrina', temperatura : 30 }
];

var kelvin = cidades.map( function( elem ) {
	return elem.temperatura + 273;
});

for(var a=0; a<kelvin.length; a++) {
	document.write(kelvin[a] + 'º K<br>');
}

// Função "kelvin" utilizando Sintaxe ES6
var kelvin = cidades.map( ( elem ) => elem.temperatura + 273 );
```

Filter: a função "filter" é utilizada remover elementos indesejados com base em alguma(s) condição(ões). Ela percorre o *array* da esquerda para a direita invocando uma função de retorno para cada elemento percorrido. O valor retornado é booleano, e indica se o elemento percorrido será mantido no novo *array* ou descartado. Após todos os elementos serem percorridos, a função *filter()* retorna o novo *array* com todos os elementos que retornaram verdadeiro.

Protótipo

```
array.filter( function( elem, index, array ) {
   ...
}, thisArg );
```

Gil Eduardo de Andrade

Parâmetro	**Significado**
elem	Valor do elemento
index	Índice em cada iteração, da esquerda para a direita
array	*Array* original invocando o método
thisArg	(opcional) Objeto que será referenciado como *this* no *callback*

Exemplo de Codificação

```
// Exemplo utilizando Array Simples
function aprovado(arr) {
	return arr >= 7;
}
var notas = [10, 5, 6, 3, 9];
var ap = notas.filter(aprovado)

for(var a=0; a<ap.length; a++) {
	document.write(ap[a] + '<br>');
}

// Exemplo utilizando Array de Objetos
function aprovado(obj) {
	return obj.nota >= 7;
}

var alunos = [
	{nome : 'Maria', nota : 10},
	{nome : 'Rodrigo', nota : 5},
	{nome : 'Gabriela', nota : 3},
	{nome : 'Bruno', nota : 7}
]

var ap = alunos.filter(aprovado);

for(var a=0; a<ap.length; a++) {
	document.write(ap[a].nome + '<br>');
}
```

Reduce: a função “reduce” é utilizada para computar um valor cumulativo ou concatenado, tendo como base todos os elementos de um *array*. Ela o percorre da esquerda para a direita invocando uma função de retorno para cada elemento. O cálculo retornado é o valor acumulado e passado a cada nova chamada da função (*callback*). Após todos os elementos serem percorridos, a função *reduce()* retorna o valor acumulado/concatenado.

Protótipo

```
array.reduce ( function ( prevVal, elem, index, array ) {
    ...
}, initialValue );
```

Parâmetro	Significado
preVal	Valor acumulado e retornado em cada iteração
elem	Valor do elemento
index	Índice em cada iteração, da esquerda para a direita
Array	*Array* original invocando o método
initalValue	(opcional) Objeto usado como primeiro argumento na primeira iteração.

Exemplo de Codificação

```
// Exemplo Utilizando Simples
var titulos = [4, 2, 1, 5];

var soma = titulos.reduce( function( prevVal, elem ) {
        return prevVal + elem;
}, 0);

document.write(soma);

// Função "soma" utilizando Sintaxe ES6
var soma = titulos.reduce( ( prevVal, elem ) => prevVal + elem, 0);

// Exemplo Utilizando Array de Objetos
var selecoes = [
        { pais : 'Alemanha', titulos : 4 },
        { pais : 'França', titulos : 2 },
        { pais : 'Espanha', titulos : 1 },
        { pais : 'Brasil', titulos : 5 }
];

var soma = selecoes.reduce( function( prevVal, elem ) {
        return prevVal + elem.titulos;
}, 0);

document.write(soma);

// Função "soma" utilizando Sintaxe ES6
```

```
var soma = selecoes.reduce( ( prevVal, elem ) => prevVal + elem.titulos, 0);
```

<u>Observação</u>: As funções de *array*: *map()*, *filter()* e *reduce()* não tem como objetivo substituir as codificações que envolvem laço de repetição, estas ainda podem e devem ser aplicadas em diversas situações como, por exemplo, quando é preciso parar a iteração se determinada(s) condição(ões) for(em) atendida(s).

VARIÁVEIS: VAR / LET / CONST

O que diferencia as várias declaradas como ***var*** ou ***let*** é o escopo de aplicação das duas, ou seja, a variável ***var*** possui um escopo de função, enquanto a variáveis ***let*** (ES6) possui um escopo de bloco. As variáveis declaradas como ***const*** funcionam de maneira similar ao ***let***, contudo seus valores não podem ser reatribuidos.

Var: em JavaScript, toda variável declarada como ***var*** é "elevada" (*hoisting*) até o topo do contexto de execução. Em outras palavras, uma variável dentro de uma ***function***, é elevada (*hoisting*) para o topo do código desta ***function***.

<u>Exemplo de Codificação</u>

```
// Exemplo var – Com Hoisting
var exibir = function() {
      msg = 'Ainda não fui declarada';
      document.write(msg);
      var msg;
}
exibir();
```

Observe que na primeira e segunda linha do código, respectivamente, atribuímos um valor a variável ***msg*** e exibimos o seu conteúdo, mas só vamos efetuar a sua declaração a terceira linha. Isso é possível porque o *JavaScript* eleva (*hoisting*) a declaração da variável para início da codificação. Portanto, devido ao *hoisting*, é possível usar uma variável antes mesmo dela ter sido declarada, já que em tempo de execução ela será elevada.

```
// Exemplo "var" vs "let"
function x() {
      var n = 1;
      if (true) {
            var n = 2;
            document.write(n); //=2
      }
```

```
        document.write(n); //=2
}

x();
```

Observe que mesmo declarando duas variáveis “n” em escopos diferentes, ao atribuirmos o valor “2” a segunda declaração, ambas as variáveis recebem esse valor, ou seja, perderíamos o valor armazenado anteriormente na declaração da primeira variável “n”. Como o exemplo é simples, a dinâmica apresentada pode parecer pouco útil, mas em aplicações maiores a probabilidade de declararmos variáveis do tipo ***var*** e com o mesmo nome aumenta, gerando resultados inesperados.

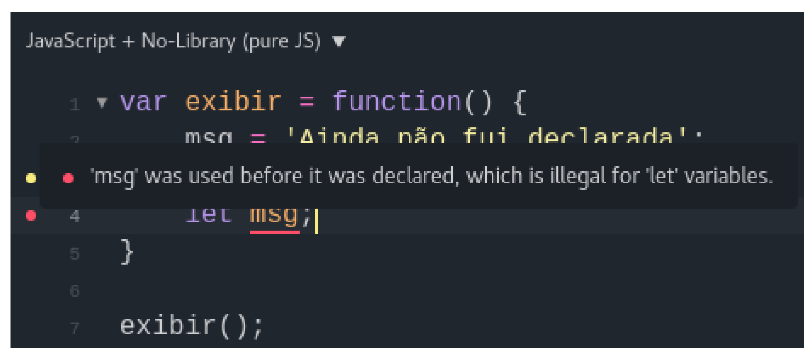
Let: a partir da versão JavaScript ES6 podemos utilizar a palavra-chave ***let***. A diferença entre elas é que quando utilizamos ***let***, estamos atribuindo escopo de bloco à variável criada e, portanto, não ocorre o *hoisting*.

Exemplo de Codificação

```
// Exemplo let – Sem Hoisting
var exibir = function() {
        msg = 'Ainda não fui declarada';
        document.write(msg);
        let msg;
}

exibir();
```

Observe que a codificação não irá funcionar como no exemplo utilizando ***var***, até por isso uma mensagem de erro é exibida, como mostrado na imagem a seguir.




```
// Exemplo “let” vs “var”
function x() {
        let n = 1;
```

Gil Eduardo de Andrade

```
        if (true) {
                let n = 2;
                document.write(n); //=2
        }

        document.write(n); //=1
}

x();
```

Const: embora o ***let*** garanta o escopo, existe a possibilidade de declararmos uma variável com ***let*** e ela ser ***undefined***.

Exemplo de Codificação

```
// Exemplo let – undefined
var exibir = function() {
        let msg;
        document.write(msg);
}

exibir();
```

Existem codificações onde é preciso garantir que uma variável foi inicializada com algum valor (diferente de ***undefined***). Para termos essa garantia, utilizamos as chamadas constantes, utilizando a declaração via ***const***. Assim como as variáveis ***let***, constantes também possuem escopo de bloco, contudo elas devem ser sempre inicializadas no momento da sua declaração.

```
// Exemplo const
var exibir = function(){
        const msg = 'Sou uma constante!';
        document.write(msg)
}();
```

Observe que a ao declararmos a constante "msg" já atribuímos um valor, para que a codificação não apresente erro. Veja também, que é possível invocar a função "exibir" colocando "()" ao seu final, o que permite abreviar a linha de código "exibir()" utilizada nos exemplos anteriores.

FUNÇÃO E PARÂMETROS DE FUNÇÃO

A nova versão JavaScript (ECMAScript 6 ou ES6) apresenta pequenas alterações em relação a

parametrização de funções que trazem benefícios (facilidades) ao desenvolvedor. Dentre as quais temos:

Default Parameters: possibilita ao desenvolvedor definir valores ***default*** para os parâmetros das funções.

Exemplo de Codificação

```
// Exemplo – Padrão ES5
var multiplicar = function(x, y) {
	y = y | 1;
	x = x | 1;
	return x * y;
}

document.write('Resultado: ' + multiplicar() + '<br>');
document.write('Resultado: ' + multiplicar(5) + '<br>');
document.write('Resultado: ' + multiplicar(5, 3) + '<br>');

// Exemplo – Padrão ES6
var multiplicar = function(x=1, y=1) {
	return x * y;
}

document.write('Resultado: ' + multiplicar() + '<br>');
document.write('Resultado: ' + multiplicar(5) + '<br>');
document.write('Resultado: ' + multiplicar(5, 3) + '<br>');
```

Observe que na sintaxe do segundo exemplo (ES6) é possível informar valores ***default*** para os parâmetros da função no momento em que elas são declaradas, sem a necessidade de linhas adicionais dentro da própria função (ES5).

Arrow Function: possibilita diminuir a escrita do código. Através dele podemos suprimir as escritas ***function*** e ***return***. Veja o resultado do uso do conceito de "***arrow function***" no exemplo anterior para ***default parameters***.

```
// Exemplo – Parâmetro default + Arrow Function
var multiplicar = (x=1, y=1) => x * y;

document.write('Resultado: ' + multiplicar() + '<br>');
document.write('Resultado: ' + multiplicar(5) + '<br>');
document.write('Resultado: ' + multiplicar(5, 3) + '<br>');
```

Arguments: a versão ***ES5*** do JavaScript possui o recurso ***arguments*** que permite obter todo os parâmetros passados a uma função, sem a necessidade de que estes sejam declarados.

Exemplo de Codificação

```
// Arguments – Padrão ES5
var media = function() {
	let result = 0;
	for (let i=0; i < arguments.length; i++) {
		result += arguments[i];
	}
	return result/arguments.length;
}

var r = media(10, 6, 8, 4);
document.write(r);
```

Observe que mesmo sem definirmos os parâmetros que serão recebidos pela função "media" é possível passar e obter valores através do recurso ***arguments***. Contudo, esse recurso apresenta algumas limitações, por exemplo, todos os parâmetros são atribuídos ao ***arguments*** (que não é exatamente um array) e não é possível diferenciá-los. Devido essas restrição o recurso ***Rest Parameters*** foi adicionado a versão ***ES6*** do JavaScript. Veja a mesma função "media" reescrita com o uso do *Rest Parameters.*

Rest Parameters: a versão ***ES6*** disponibiliza o recurso ***rest parameters*** que permite obter todos os em formato *array*.

Exemplo de Codificação

```
// Rest Parameters – Padrão ES6
var media = function(...pars) {
	let result = 0;
	pars.forEach((valor) => {
		result += valor;
	})
	return result/pars.length;
}

var r = media(10, 6, 8, 4);
document.write(r);

// Rest Parameters ES6 + Reduce + Arrow Function
const media = function(...pars) {
```

```
        let soma = pars.reduce((preVal, elem) => preVal + elem, 0);
        return soma/pars.length;
}

var r = media(10, 6, 8, 4);
document.write(r);
```

Destructuring Assignment: permite que um novo formato para a declaração das variáveis seja utilizado, onde é possível, já no momento da declaração, extrair valores de objetos e *arrays* via recurso denominado ***destructuring assignment***.

Exemplo de Codificação

```
// Destructuring Assignment – Array
const [a, b] = [10, 20];
document.write(a + " " + b);
```

Observe que ao declararmos as constantes “a e b” já atribuímos os valores “10 e 20” vindos do *array*.

```
// Destructuring Assignment – Array + Rest
const [a, b, ...rest] = [10, 20, 30, 40, 50];
document.write(a + " " + b + " " + rest);
```

Observe que ao declararmos as constantes “a, b e rest”, “a e b” recebem os valores “10 e 20” e o “rest” recebe todos os outros valores (“30, 40, 50”) do *array*.

```
// Destructuring Assignment – Objeto
const aluno = { nome : 'Matheus', idade : 17 };
const {nome, idade} = aluno;

document.write(nome + " / " + idade + " anos");
```

Observe que ao declararmos as constantes “nome, idade”, elas recebem, respectivamente, os valores ‘Matheus’ e ‘17’ do array de objeto. Essa abordagem é permitida desde que as constantes (const{nome, idade}) tenham o mesmo nome do identificador do atributo do objeto, nesse caso “nome” e “idade”.