

Valderi Leithardt, Dr.

IPW (Class notes for 2 weeks)

Summary

- Review of previous class *
- Objects
- Functions
- Arrays
- Exercises

Operators

→ Estruturas de controle

if ... else

....

...

switch ... case

...

.....

while

....

..

do ... while

....

..

for

..

...

for ... in

Instruções e declarações

→ **Let**

Declara uma variável local no escopo do bloco atual, opcionalmente iniciando-a com um valor.

Definição

```
let var1 [= value1] [, var2 [= value2]] [, ..., varN [= valueN]];
```

Descrição

→ let permite que você declare variáveis limitando seu escopo no bloco, instrução, ou em uma expressão na qual ela é usada.

→ Isso é inverso da keyword var (en-US), que define uma variável globalmente ou no escopo inteiro de uma função, independentemente do escopo de bloco.

Instruções e declarações

→ **CONST**

- Constantes possuem escopo de bloco, semelhantes às variáveis declaradas usando a palavra-chave **let**.
- O valor de uma constante não pode ser alterado por uma atribuição, e ela não pode ser redeclarada.

Sintaxe

```
const name1 = value1 [, name2 = value2 [, ... [, nameN = valueN]]];
```

Descrição

- A declaração `const` cria uma variável cujo o valor é fixo, ou seja, uma constante somente leitura. Isso não significa que o valor é imutável, apenas que a variável constante não pode ser alterada ou retribuída.
- Esta declaração cria uma constante que pode pertencer tanto ao escopo global (na janela ou objeto) quanto ao local do bloco em que é declarada. Constantes globais não se tornam propriedades do objeto `window`, diferente da criação de variáveis com `var`.

Objects

- Diferente de uma variável, um objeto pode conter diversos valores e/de tipos diferentes armazenados nele (atributos) e também possuir funções que operem sobre esses valores (métodos). Tanto os atributos, quanto os métodos, são chamados de propriedades do objeto.

→ Criando objetos usando a sintaxe literal de objeto

```
const pessoa = {  
  nome: 'testeNome',  
  sobrenome: 'testeSobrenome'  
};
```

→ Usando a palavra-chave 'new' com a função construtora Object integrada

```
const pessoa = new Object();
```

Objects

→ Adicionar propriedades

```
pessoa.nome = 'testeNome'; pessoa.sobrenome = 'testeSobrenome';
```

→ Usando a palavra-chave 'new' com a função construtora Object integrada

```
const pessoa = new Object();
```

→ Para adicionar propriedades a esse objeto, temos de fazer algo semelhante a isso:

```
pessoa.nome = 'testeNome'; pessoa.sobrenome = 'testeSobrenome';
```

→ Usando 'new' com uma função construtora definida pelo usuário

```
function Pessoa(primNome, sbrNome)
{ this.nome = primNome;
  this.sobrenome = sbrNome; }
```

Objects

→ Para criar um objeto 'Pessoa', basta fazer isso:

```
const pessoaUm = new Pessoa('testeNomeUm', 'testeSobrenomeUm');  
const pessoaDois = new Pessoa('testeNomeDois', 'testeSobrenomeDois');
```

→ Usar `Object.create()` para criar objetos

» organização representada por `objOrg`

```
const objOrg = { empresa: 'ABC Corp' };
```

→ Consultar os funcionários dessa organização.

» Querer todos os objetos funcionario.

```
const funcionario = Object.create(objOrg, { nome: { valor: 'FuncionarioUm' } });  
console.log(funcionario); // { empresa: "ABC Corp" }  
console.log(funcionario.nome); // "FuncionarioUm"
```


Objects

→ Usar `Object.assign()` para criar objetos

» Considere que você tenha dois objetos

```
const objOrg = { empresa: 'ABC Corp' }  
const objCarro = { nomeCarro: 'Ford' }
```

» Agora, você quer um objeto funcionario de 'ABC Corp' que dirija um carro 'Ford'.
Você pode fazer isso com a ajuda de `Object.assign`:

```
const funcionario = Object.assign({}, objOrg, objCarro);
```

» Nesse momento, você tem um objeto **funcionario**
que possui **empresa** e **nomeCarro** como suas propriedades

```
console.log(funcionario); // { nomeCarro: "Ford", empresa: "ABC Corp" }
```

Objects

→ Usar as classes da ES6 para criar objetos

» o uso desse método é similar ao uso de 'new' com a função construtora definida pelo usuário.
As funções construtoras foram substituídas por classes já que têm o suporte das especificações da ES6.

```
class Pessoa {  
  constructor(primeiroNome, sbrNome) {  
    this.nome = primeiroNome;  
    this.sobrenome = sbrNome;  
  }  
}
```

```
const pessoa = new Pessoa('testeNome', 'testeSobrenome');  
console.log(pessoa.nome); // testeNome  
console.log(pessoa.sobrenome); // testeSobrenome
```

Functions

--» Uma função é criada por meio de uma expressão que se inicia com a palavra-chave `function`.

--» Funções podem receber uma série de parâmetros (nesse caso, somente `x`) e um "corpo", contendo as declarações que serão executadas quando a função for invocada.

--» O "corpo" da função deve estar sempre envolvido por chaves, mesmo quando for formado por apenas uma simples declaração (como no exemplo anterior).

--» Uma função pode receber múltiplos parâmetros ou nenhum parâmetro.

Functions

Existem cinco tipos de definições de funções:

- Functions declaration (Função de declaração)
- Functions expression (Função de expressão)
- Arrow Functions (Função de flecha)
- Functions constructor (Função construtora)
- Generator Functions (Função gerador)

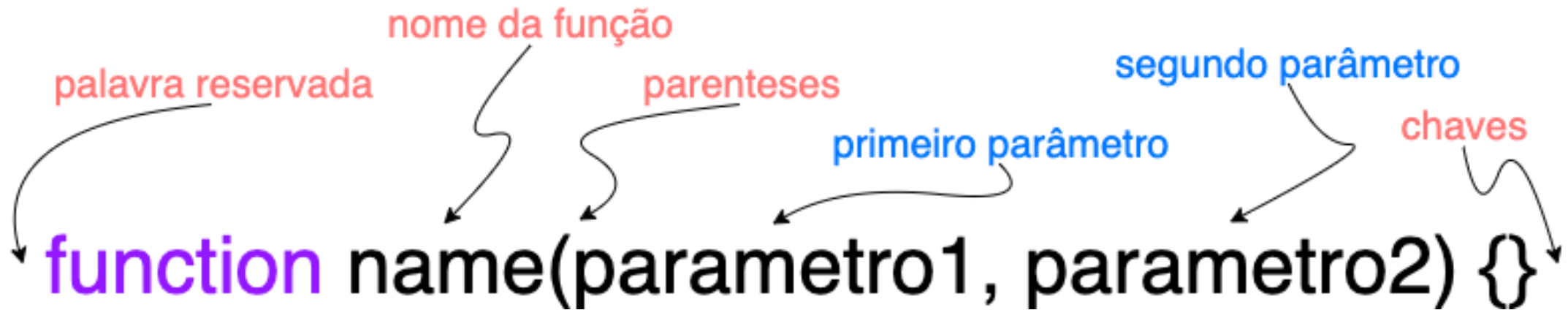
--» As definições mais comuns e similares: declaration e expression.

Functions



--» Estrutura mais simples, No entanto, obrigatória para as **functions declaration**.

Functions



A diagram illustrating the components of a function definition. The code `function name(parametro1, parametro2) {}` is shown. Above the code, several labels in red and blue text are connected to parts of the code by arrows. The labels are: 'palavra reservada' (red) pointing to 'function'; 'nome da função' (red) pointing to 'name'; 'parenteses' (red) pointing to the opening parenthesis '('; 'primeiro parâmetro' (blue) pointing to 'parametro1'; 'segundo parâmetro' (blue) pointing to 'parametro2'; and 'chaves' (red) pointing to the closing curly brace '}'.

palavra reservada nome da função parenteses primeiro parâmetro segundo parâmetro chaves

`function name(parametro1, parametro2) {}`

--» Também podemos definir parâmetros opcionais separados por vírgula:

Functions

Algumas curiosidades:

- > Em JavaScript podemos declarar funções dentro de funções.
- ➔ Uma função declarada dentro de outra, apenas irá viver durante o escopo da função pai, ou seja, a função mensagem apenas existe no escopo/bloco da função Hello.
- ➔ Para invocar uma função utilizamos o seu nome seguido por parenteses ().
- ➔ Para alimentar algum parâmetro, adicionamos o valor dentro dos parenteses ('Mundo'), onde a ordem dos parâmetros irá influenciar, ou seja, se uma função recebe dois parâmetros:
- ➔ `function ola(nome, sobrenome)`, ao chamá-la precisamos tomar cuidado com a ordem dos mesmos: `Hello('Mundo', 'IPW')` é diferente de `Hello('IPW', 'Mundo')`

Functions

The diagram illustrates the syntax of a function expression with the following components and annotations:

- const**: Annotate as "palavra reservada" (reserved word).
- name**: Annotate as "nome da variável" (variable name).
- =**: Annotate as "palavra reservada" (reserved word).
- function**: Annotate as "parenteses" (parentheses).
- parameter1**: Annotate as "primeiro parâmetro" (first parameter).
- parameter2**: Annotate as "segundo parâmetro" (second parameter).
- { }**: Annotate as "chaves" (braces).

```
const name = function(parameter1, parameter2) { }
```

--» **Functions expression** (Função de expressão):

- Atribuir uma função à uma variável pode ser muito útil, por exemplo:
- Assim pode-se definir a função exatamente onde ela precisa ser chamada.

Functions

The diagram illustrates the syntax of an arrow function: `const name = (parametro1, parametro2) => {}`. Hand-drawn arrows point from labels to specific parts of the code: 'palavra reservada' (reserved word) points to 'const'; 'nome da variável' (variable name) points to 'name'; 'parenteses' (parentheses) points to the opening '('; 'primeiro parâmetro' (first parameter) points to 'parametro1'; 'segundo parâmetro' (second parameter) points to 'parametro2'; 'arrow simbolo' (arrow symbol) points to the '=>' symbol; and 'chaves' (braces) points to the closing '}'.

--» **arrow functions** são simplificações para as functions expression:

- um dos motivos da criação das funções de flecha é facilitar a criação e utilização de funções em JavaScript, ou seja, elas permitem a criação de funções de maneira resumida.

```
function nomeCompleto(nome, sobrenome) {  
  return `${nome} ${sobrenome}`  
}
```

Functions

--» Functions constructor

→ As funções construtoras são declaradas e definidas como qualquer outra expression ou declaration, a forma de utilizar é a mesmo, a diferença está mais no caso de uso e o que ela retorna.

→ Uma pequena observação é que normalmente o nome de funções construtoras começa com a primeira letra maiúscula, por exemplo:

```
function Pessoa() {}
```

Função construtora que irá criar um objeto Pessoa --> A principal diferença entre a função construtora está na maneira como ela é invocada, enquanto as demais apenas precisam ser nomeadas e utilizar os parenteses:

```
function ola() {}  
ola()  
  
const ola = function() {}  
ola()
```

As funções construtoras precisam ser invocadas com a palavra reservada new:

```
const p = new Pessoa()
```

Functions

--» Generator Functions

→ A definição e declaração da mesma é muito semelhante as funções de expressão e declaração, uma pequena diferença está na adição de um * na palavra reservada function, ou seja, function* :

```
function* ola(p1, p2) {}
```

As demais regras se aplicam para a mesma, onde os parâmetros são opcionais e separados por vírgula e o corpo da função fica dentro das chaves.

```
function ola() {  
  console.log('Olá')  
  console.log('Turma')  
  console.log('IPW')  
}  
ola()
```

Agora temos uma nova palavra reservada, a yield, essa palavra indica quais são os passos e onde a função deve ir parando sua execução, ou seja, cada yield é um ponto de interrupção da função.

```
function* ola() {  
  yield 'Olá'  
  yield 'Turma'  
  yield 'IPW'  
}  
  
for (const n of ola()) {  
  console.log(n)  
}
```

Functions

--» **yield**

A palavra-chave `yield` é usada para pausar e resumir uma generator function (function* or generator function legada (en-US)).

Expressão

Define o valor que retorna de uma generator function via o protocolo iterator. Se omitido, será retornado `undefined`.

Descrição

A palavra-chave `yield` pausa a execução de uma generator function e o valor da expressão em frente a palavra-chave `yield` é retornado para a chamada do generator. Ele pode ser considerado uma versão da palavra-chave `return` para o generator.

→ **yield***

A expressão `yield*` é usada para delegar para outro objeto generator ou iterable.

Descrição

A expressão `yield*` itera sobre a operação e `yields` cada valor retornado por ele.

O valor da expressão `yield*` sozinha é o valor retornado pelo iterator quando ele for fechado (i.e., quando `done` é `true`).

Arrays

- Os Arrays são pares do tipo inteiro-valor para se mapear valores a partir de um índice numérico.
- Em JavaScript os Arrays são objetos com métodos próprios.
- Um objeto do tipo Array serve para se guardar uma coleção de itens em uma única variável.

Exemplo:

```
var arr = new Array();  
// Por ser um objeto podemos usar o "new" em sua criação  
var arr = new Array(elem1,elem2, ... ,elemN);  
// Dessa forma criamos um array já iniciado com elementos.  
var arr = [1,2,3,4];  
// outra forma é iniciar um array com elementos sem usar o "new".  
var arr = new Array(4);  
// Dessa forma criamos um array vazio de 4 posições.
```

Arrays

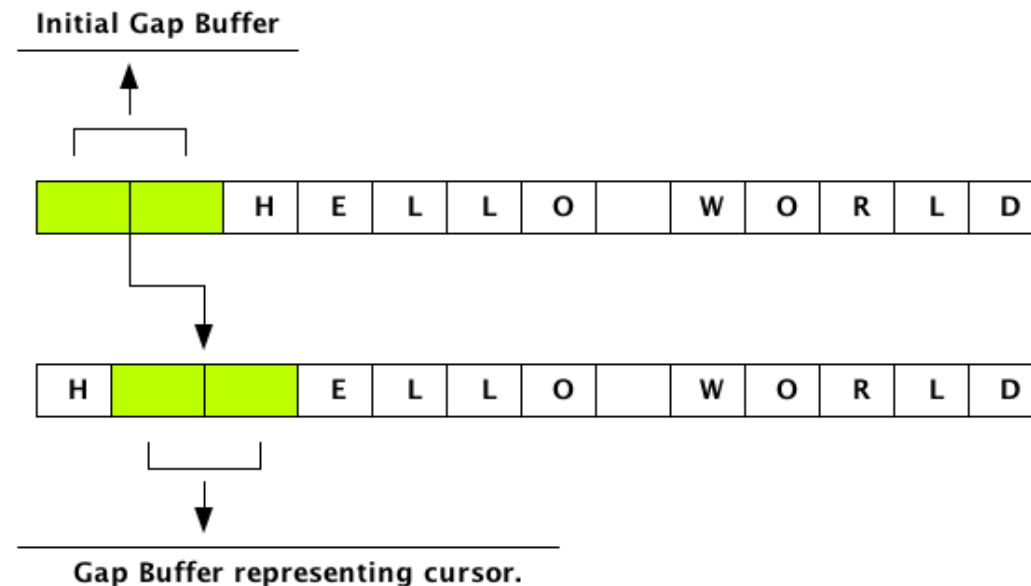
- Para acessar as variáveis dentro de um array basta usar o nome do array e o índice da variável que se deseja acessar.
- Do mesmo modo, pode-se fazer atribuições ou simplesmente ler o conteúdo da posição.
- Em JavaScript os **arrays podem conter valores de tipos diferentes** sem nenhum problema; podemos colocar em um mesmo array inteiros, strings, booleanos e qualquer objeto que se desejar.

Exemplo:

```
arr[0] = "Até mais e obrigado pelos peixes";  
arr[1] = 42;  
document.write(arr[1]);  
//imprime o conteúdo de arr[1]
```

Arrays

- Array é uma estrutura de dados, todos do mesmo tipo. Vetores (1D) e Matrizes (2D, 3D, ...) são exemplos de arrays.
- Arrays são geralmente descritas como "lista de objetos"; elas são basicamente objetos que contem múltiplos valores armazenados em uma lista.
- Um objeto array pode ser armazenado em variáveis e ser tratado de forma muito similar a qualquer outro tipo de valor, a diferença está em podermos acessar cada valor dentro da lista individualmente.



Arrays

Arrays são construídos através de um construtor e possuem tamanho dinâmico:

```
var nomes = newArray();  
//var nomes = [];  
nomes[0] = "Joao";  
nomes[1] = "Maria";  
nomes.push("Jose");
```

[Documentação W3S](#)

Arrays

```
var frutas = ["Maçã", "Banana"];
```

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

Reference [mozilla](#)

```
const fruits = ["Banana", "Orange", "Apple"];
```

```
fruits instanceof Array;
```

Reference [W3S](#)

Exercises

* Concluir exercícios da aula anterior.

- https://eloquentjavascript.net/03_functions.html Exercícios final do capítulo
- https://eloquentjavascript.net/04_data.html Exercícios final do capítulo

References

- <https://eloquentjavascript.net/>
- <https://developer.mozilla.org/pt-BR/docs/Web/JavaScript/Reference>
- <https://www.freecodecamp.org/news/author/freecodecamp/>
- <https://www.devmedia.com.br/javascript-arrays/4079>
- <https://github.com/braziljs/eloquente-javascript/tree/master>
- Adaptações e modificações com base no material disponibilizado pelo Professor Luís Falcão, acesso online em: <https://github.com/isel-leic-ipw/>

Valderi Leithardt, Dr.

Professor IPW

valderi.leithardt@isel.pt