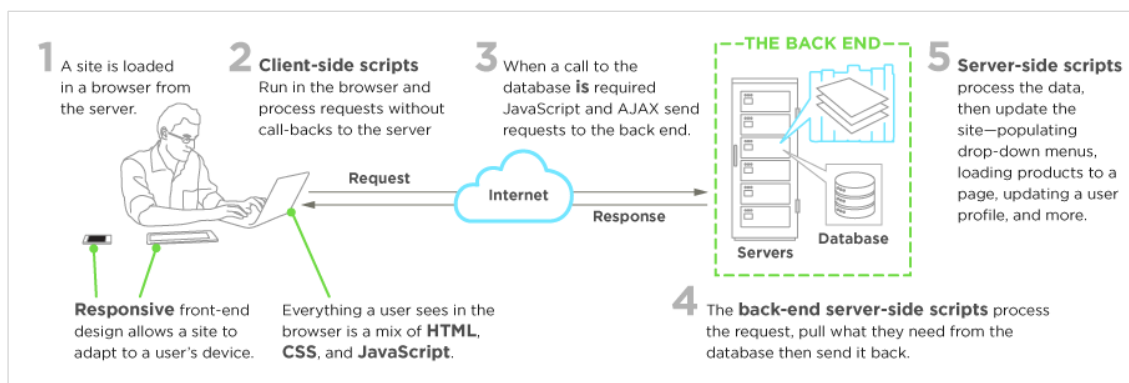


JavaScript

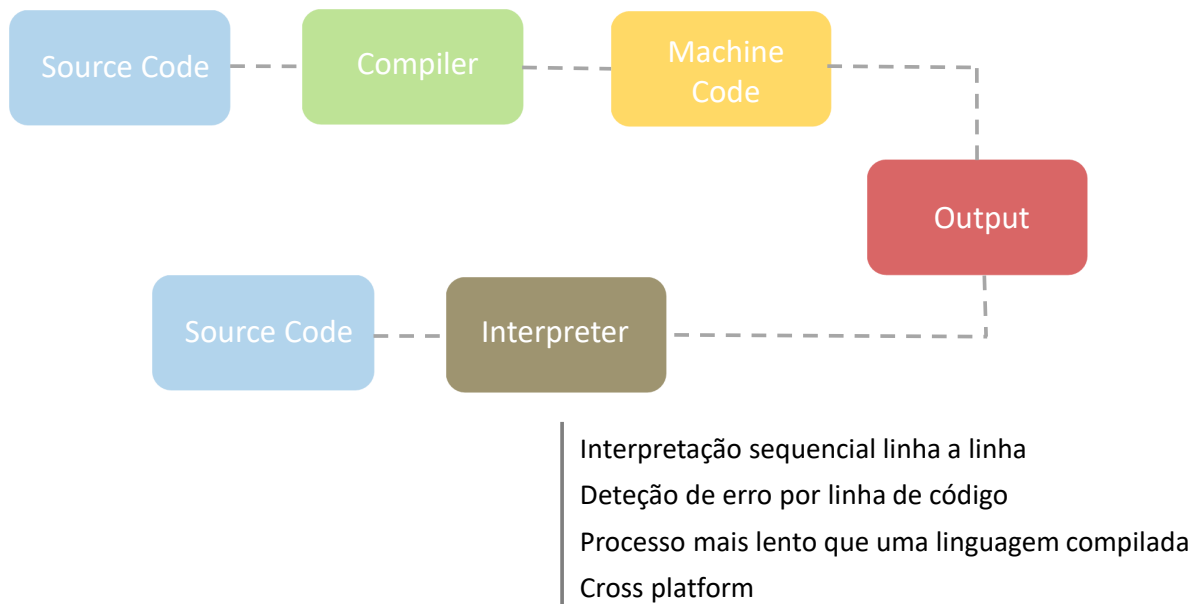
Linguagens de script

- Linguagem de programação integrada em outro programa/código
 - Linguagens interpretadas
 - Não necessitam de compilador
 - Javascript; PHP; ...



<https://www.upwork.com/hiring/development/how-scripting-languages-work/>

■ Linguagens Interpretadas vs. Linguagens Compiladas



JavaScript

■ Scripting language

“JavaScript is **THE** scripting language of the Web.”

<http://www.w3schools.com/js/default.asp>

- Começou por ser exclusivamente uma **client-side scripting language**
 - Interpretada diretamente pelo *browser (on the fly)*, não necessita de ser compilada
- Atualmente também utilizada no lado do servidor (*server-side*)
 - *Node.js*
- Executado
 - Após o download
 - Como resposta a um evento
- Permite:
 - Geração dinâmica de conteúdo / Efeitos
 - Melhorar a experiência do utilizador:
 - Interactividade
 - Resposta a eventos, validação de dados, ...
 - Gerir a comunicação com o servidor
 -



<https://betterdocs.co/top-scripting-languages/>

Inserção de scripts

Embedded Script

- `<script> ... </script>`
 - O *script* pode ser colocado no *head* ou no *body*
 - preferencialmente, para maximizar a performance, o *script* deve ser colocado no final do *body* não influenciando assim o tratamento dos restantes elementos HTML
 - O *script* pode ser executado quando é efetuado o **download** do *.html (sem controlo por um evento)

```
<script>
  init();

  function init(){
    alert('Script executado automaticamente');
  }
</script>
```

This page says
Script executado automaticamente

OK

Embedded Script

(diretamente definido entre as tags `<script>`)

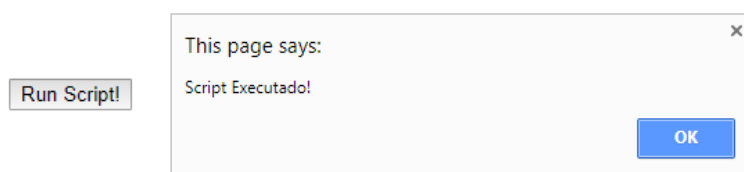
Embedded Script

- **<script> ... </script>**
 - o *script* pode ser executado como resposta a um evento, e.g. *onclick*

```
<button onclick="scriptFunction()">Run Script!</button>

<script>
  function scriptFunction(){
    alert('Script Executado!');
  }
</script>
```

Embedded Script
(diretamente definido entre as tags <script>)



Script Externo (*.js)

- **<script src="*.js"> ... </script>**

```
<body>

  <button onclick="scriptFunction()">Run Script!</button>

  <script src="external.js"></script>

</body>
```

Executa a função `scriptFunction()`
declarada no ficheiro `external.js`

Ligação ao **ficheiro externo** na tag
<script> atributo **src**

```
function scriptFunction(){
  alert('Script Executado!');
}
```

external.js

■ Scripts

Embebido no HTML

Ficheiros Externos
(extensão *.js)

■ Controlo da execução do script

Executado após o
download

Executado só após a
ocorrência de um evento

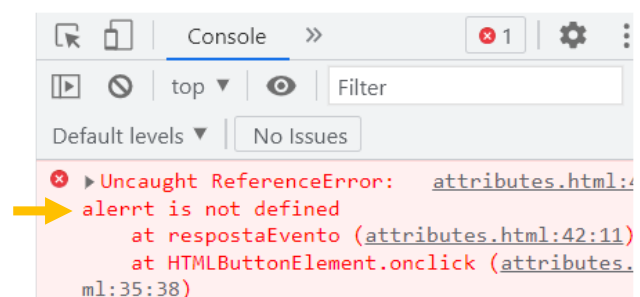
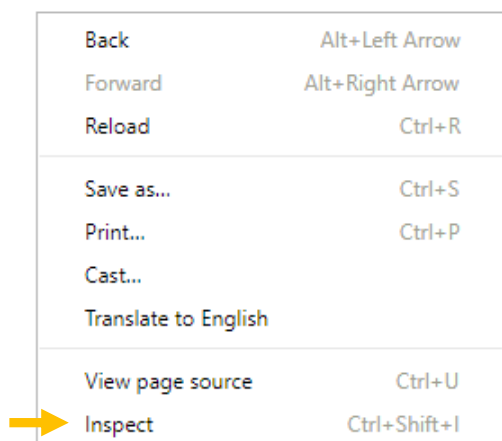
Chamada a uma função, em que a
função pode ser:

- Criada pelo utilizador
- Nativa

Browser (development tools)

■ Browser Console

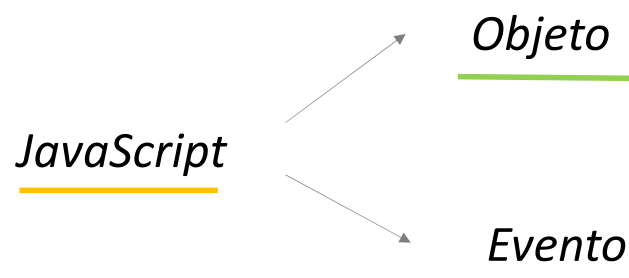
- Permite obter informação sobre o valor das variáveis, erros, *warnings*, *requests*, ...
- Muito importante para efetuar o *debugging*/controlo completo dos scripts
 - Linguagem interpretada (sequencial linha a linha)



Conceitos Chave

JS

Conceitos Chave



Conceitos Chave

Objeto

- armazenar dados, estruturação da aplicação, código mais limpo/modular
 - Identidade
 - Propriedades
 - Métodos

```
<script>
  var hotel = {

    name: 'Coimbra',
    rooms: 20,
    booked: 15,
    gym: true,
    roomTypes: ['single', 'double', 'suite'],

    checkAvailability: function () {
      return this.rooms - this.booked;
    }

  }
</script>
```



Javascript

Evento

- Ação que pode ser detetada pelo *JavaScript* e que provoca uma execução específica:

- Chamada de uma função
- A função **só é executada após a ocorrência** do respetivo evento

- Exemplos:

| Evento | É disparado... |
|-----------|---|
| click | quando é pressionado e liberado o botão primário do mouse, trackpad, etc. |
| mousemove | sempre que o cursor do mouse se move. |
| mouseover | quando o cursor do mouse é movido para sobre algum elemento. |
| mouseout | quando o cursor do mouse se move para fora dos limites de um elemento. |
| dblclick | quando acontece um clique duplo com o mouse, trackpad, etc. |

Sintaxe

JS

Sintaxe JS

- *case sensitive*.
- `//` símbolo do comentário
 - `/*` comentário para múltiplas linhas `*/`
- Um *script* é composto por um conjunto de *statements/expressions*

```
function Calculo(formulario)
{
    if(confirm("Confirma?"))
        formulario.result.value = eval(formulario.expr.value);
    else
        alert("Novos dados");
}
```

- Os *code blocks* (conjuntos de instruções) são delimitados por `{ ... }`
 - Elementos fundamentais para a estruturação do código

Sintaxe JS

■ *Expression*

- a sua execução origina sempre um valor:

- *numérico*
- *string*
- *boolean*

- podem ser parte de *statements*

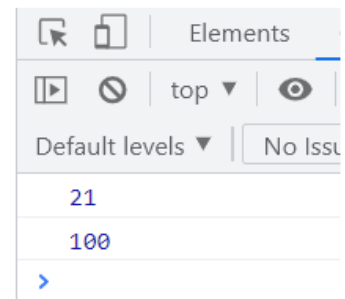
```
var sum;
var a=2;

function Modify(i){
  return i*=10
}

sum = 20;  // assign 20
sum++;    // increment

a=Modify(10); // modifies the value of a

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



Sintaxe JS

■ *Statement*

- a sua execução **produz uma ação mas não gera um valor imediato**

- podem conter *expressions*
- são executados isoladamente pela ordem em que são escritos

```
var sum; // statement
var a=2  // statement + assignment expression

function Modify(i){    //function declaration statement
  return i*=10
}

sum = 20;
sum++;

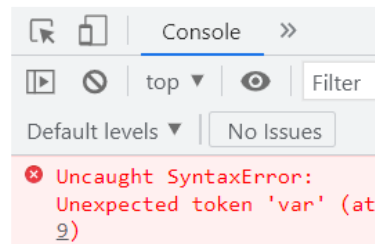
a=Modify(10);

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

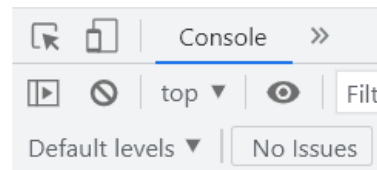
Sintaxe JS

- *Statements* não podem ser utilizados onde é esperada uma *expression*

```
var sum = var a
```



```
var a=2;  
var b = (a=5);  
console.log(b);
```



5

Sintaxe JS

- A utilização do “;” não é consensual, existem 2 perspetivas:

- “Omit Semicolon School” (*Automatic Semicolon Insertion*)

- “Add Semicolon School”

- Código mais estruturado, facilita a leitura:

- Algumas regras:

- usar sempre ; que se tratar de uma expressão *top level*

```
let x=4;
```

- não é necessário ; no final de :

- declaração de uma função *function name (...)* {...}
- if (...) {...} else {...}
- for (...) {...}
- while (...) {...}

- necessário ; quando:

- do{...} while (...);

Variáveis

JS

Variáveis

■ Declaração de variáveis

- armazenamento temporário (uma vez que após o fecho da página o browser não retém o valor atribuído à variável)
- **loosely typed**, não é necessário definir o tipo de variável uma vez que este é automaticamente assumido de acordo com a declaração (atribuição) efetuada

■ **var**

■ **let**

- A variável só é visível no bloco onde foi criada

■ **const**

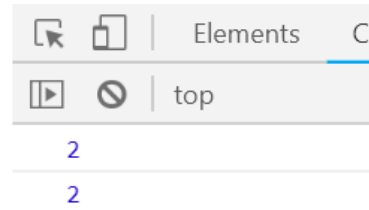
- Não permite alterar a atribuição do valor inicial (declaração).

```
var x=10;      //numeric
var x="ten"    //string
var x;
```

Scope (let)

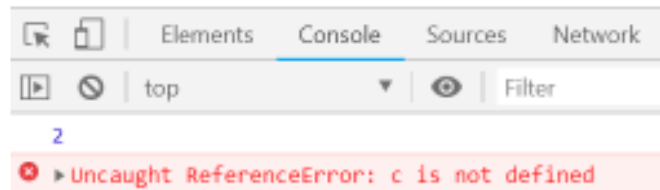
■ var (scope é a função)

```
calculateSum(2);  
  
function calculateSum (a,b = 1){  
  if (b==1)  
  {  
    var c=2;  
    console.log(c);  
  }  
  console.log(c);  
};
```



■ let (scope é o bloco {...})

```
calculateSum(2);  
  
function calculateSum (a,b = 1){  
  if (b==1)  
  {  
    let c=2;  
    console.log(c);  
  }  
  console.log(c);  
};
```

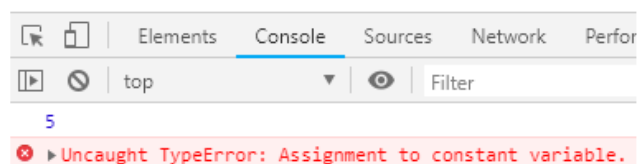


const

■ const

- declarar uma constante
 - sempre que se pretende atribuir a uma variável um valor que não é susceptível de ser alterado
 - frequentemente utilizado para a declaração de objetos (referência constante)

```
calculateSum(2);  
  
function calculateSum (a,b = 1){  
  const c = 5;  
  console.log(c);  
  c=a+b;  
  console.log(c);  
};
```



Variáveis

- Regras para definir o nome das variáveis:

```
var name = 'John';  
var age = 26;  
var isMarried = true;
```

- Significado semântico (ex: *firstName*, ...)
- *camelCase* (convenção)
- Podem começar por uma letra, por "\$" ou por *underscore* "_".
- Não podem conter espaços nem caracteres especiais (! , / \ + * = ...)
- Não podem conter *keywords* (ex: *var* ,)
- Apesar de ser possível, não se devem diferenciar as variáveis apenas com base nas minúsculas e maiúsculas (ex: *score* e *Score*).

Variáveis

- O *JavaScript* permite a declaração de variáveis tendo por base dois grandes tipos:

Primitive Types

Armazenados como dados simples
Contêm diretamente os valores que
lhes são atribuídos

Reference Types

Armazenados como objetos

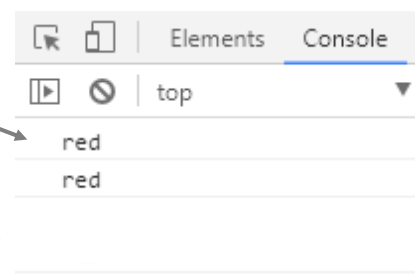
Primitive Types

JS

Primitive Types

- A variável contém diretamente o valor atribuído
 - Se for igualada a outra variável o seu valor é diretamente atribuído a essa variável
 - Apesar de partilharem o mesmo valor, as variáveis são **totalmente independentes**
 - Duas localizações de memória diferentes

```
<script>  
  var color1="red";  
  var color2=color1;  
  
  console.log(color1);  
  console.log(color2);  
  
  color1="blue";  
  
  console.log(color1);  
  console.log(color2);  
</script>
```



Primitive Types

- O JS possui 5 *primitive types*:

- number

```
var ccount=25;
```

- string

```
var name="string exemplo";
```

- boolean

```
var found=true;
```

- null

```
var obj = null;
```

- undefined

- variável sem inicialização definida

```
var data;
```

Primitive Types

- **number**

- Todos os números são representados através de *floats* de 64 bits

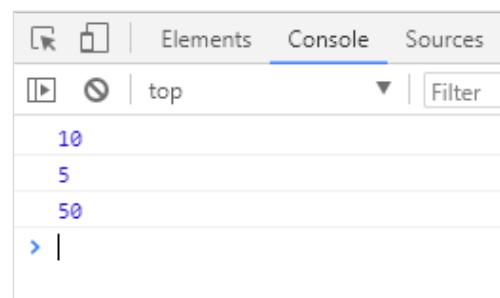
*Unlike many other programming languages, JavaScript **does not define** different types of numbers, like integers, short, long, floating-point etc.*

https://www.w3schools.com/js/js_numbers.asp

- var num1 = 50;
- var num2 = 10.5;
- var num3 = 10 * 10;

```
<script>
  var price=10;
  var quantity=5;
  var total=price*quantity;

  console.log(price);
  console.log(quantity);
  console.log(total);
</script>
```



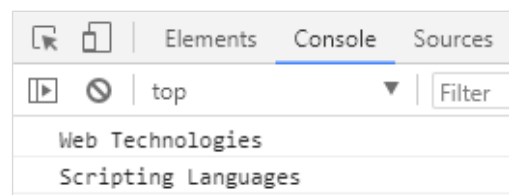
Primitive Types

■ *string*

- Cadeia de caracteres
- Declaração de uma string:
 - pode ser declarada com aspas “ ou com plica ‘, no entanto a declaração tem ser iniciada e finalizada da mesma forma
 - Quando se pretende incorporar “ ou ‘ numa string, deve declarar-se a string com o símbolo que não se pretende representar.
 - Em alternativa pode recorrer-se a uma backslash \ antes da aspa ou da plica que se pretende representar.

```
<script>
  var msg1 = "Web Technologies";
  var msg2 = 'Scripting Languages';

  console.log(msg1);
  console.log(msg2);
</script>
```

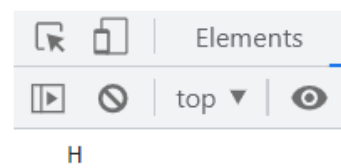


Primitive Types

■ *string*

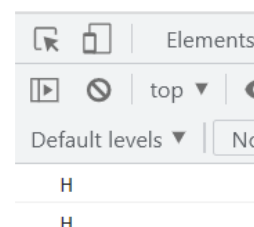
- Uma *string* permite indexação, os **índices iniciam-se em 0**:

```
var s="Hello World!"
console.log(s[0])
```



- Ao contrário de outras linguagens, ex: C, apesar de permitir indexação uma *string* não pode ser diretamente alterada

```
var s="Hello World!"
console.log(s[0])
s[0]='K'
console.log(s[0])
```



Primitive Types

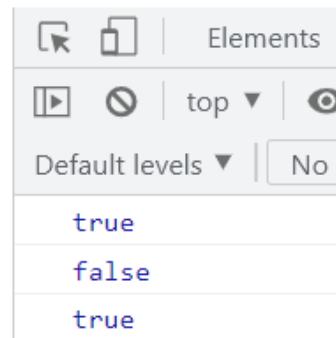
■ **boolean**

- podem assumir apenas dois valores:

- **true**

- **false**

```
var a=this;  
var b=false;  
var c=1;  
  
console.log(Boolean(a));  
console.log(Boolean(b));  
console.log(Boolean(c));
```



Reference Types (objects)

JS

Reference Types

▪ Objeto

- É uma lista não ordenada de **propriedades**, consistindo num nome e num valor.
 - Quando o valor é uma função, cria-se um método.
- Formas diferentes de criar objetos:
 - Forma Literal
 - Operador **new** + constructor **Object()**
 - *constructor* é uma função que permite a criação de um objeto com base no operador **new**
 - Através de uma **class**

Reference Types

- Ao contrário dos *Primitive Types* os **Reference Types** não guardam o objeto diretamente na variável:
 - na realidade a variável contém um **ponteiro (referência)** para a localização em memória onde o objeto existe.

```
var obj1=new Object();
```



- Quando se atribui um objeto a uma variável, na realidade essa variável armazena um ponteiro que referencia o mesmo objeto
 - De facto existe apenas um objeto, o qual está a ser referenciado (apontado) por duas variáveis.

```
var obj1=new Object();  
var obj2=obj1;
```



Reference Types

- O mesmo objeto referenciado por duas variáveis



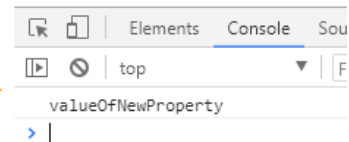
```
<script>
  var obj1=new Object();
  var obj2=obj1;
  obj1.newProperty="valueOfNewProperty";
  console.log(obj2.newProperty);
</script>
```

criado **obj1**, ponteiro para 1 objeto em memória ←

obj2 contém ponteiro para o mesmo objeto ←

adicionar nova propriedade ao objeto ←

uma vez que também é referenciado por obj2 ←



Reference Types

■ Declaração de Objetos

■ Forma Literal

■ Propriedades são formadas por:

- **identificador**
- **:valor**
- múltiplas propriedades são separadas por **virgulas**
- termina com **};**

```
const books={
  title:'Javascript',
  year: 2023
};
```

■ A ordem das propriedades é irrelevante.

■ *new* + **constructor** `Object()`

A *constructor* is useful when you want to **create multiple similar objects** with the same properties and methods.

```
const books= new Object();

books.title= 'Javascript';
books.year= 2023
```

Propriedades / Métodos

(objects)

(.)dot notation

■ Aceder a propriedades

■ *objectName.propertyName*

```
const books={  
  title:'Javascript',  
  pages:456,  
  editor: 'Packt Books'  
}
```

```
alert('Book Title: ' + books.title)
```

Book Title: Javascript

OK

■ Aceder ao editor?

```
alert('Book Editor: ' + books.editor)
```

Book Editor: Packt Books

OK

(.)dot notation

- Aceder a métodos

- *objectName.methodName()*

```
const books={
  title:'Javascript',
  pages:456,
  editor: 'Packt Books',

  showDetails: function(){
    return ('Book title: ' + this.title + '    Book pages:  ' + this.pages)
  }
}

alert(books.showDetails())
```

Book title: Javascript Book pages: 456



Reference Types

- Alterar o valor de propriedades:

```
const books={
  title:'Javascript',
  pages:456,
  editor: 'Packt Books',

  showDetails: function(){
    return ('Book title: ' + this.title + '    Book pages:  ' + this.pages)
  }
}
```

```
books.editor='Willey'
alert('Editor:  ' + books.editor)
```

Editor: Willey



- Os objetos podem ser alterados em qualquer momento:

- Propriedades podem ser alteradas/adicionadas/removidas

Reference Types

- Criar Propriedades/Métodos

```
const books={
  title:'Javascript',
  pages:456,
  editor: 'Packt Books',

  showDetails: function(){
    return ('Book title: ' + this.title + '    Book pages: ' + this.pages)
  }
}
```

```
books.chapters = 14
alert('Book Chapters: ' + books.chapters)
```

Book Chapters: 14



- Os objetos podem ser alterados em qualquer momento:
 - Propriedades podem ser alteradas/adicionadas/removidas

Reference Types

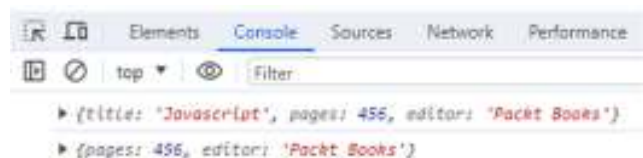
- Apagar propriedades/métodos

```
const books={
  title:'Javascript',
  pages:456,
  editor: 'Packt Books',
}

console.log(books)

delete(books.title)

console.log(books)
```



- Os objetos podem ser alterados em qualquer momento:
 - Propriedades podem ser alteradas/adicionadas/removidas

Exercício 1 (properties/methods)

Exercício 1

- Criar um objeto carro, cuja propriedades são:
 - Marca: BMW
 - Cilindrada: 2000
 - Combustível: gasoleo

- criar um método que calcule o imposto de circulação (0.05€/cc)

Exercício 1

- garanta que o valor por cc para cálculo do imposto é passado como argumento ao método `calculoluc()`

- Faça variar o valor do coeficiente/cc
 - ex: 0.04/cc

Exercício 1

- altere o valor da propriedade marca de 'BMW' para 'Mercedes', sem alterar a declaração original do objeto.
- Adicione a propriedade pintura com o valor "metalizada"

Built-in Types (Reference Types)

Built-in Types

■ Built-in types

- Criar *objects* com o constructor (keyword **new**)

- Object `const books = new Object()`

- Array `const items = new Array()`

- Date

- Error

- Function

- ...

- Os *built-in types* podem ter formas literais

- Sintaxe literal permite a criação de objetos sem utilizar o operador **new** e o respetivo *constructor*

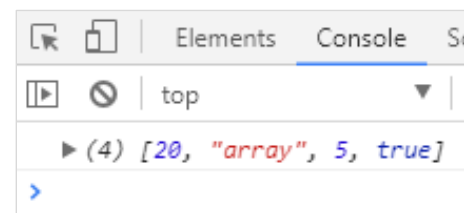
`const items = [];`

Built-in Types

■ Array

- Permite armazenar um conjunto de valores relacionados
- Ao contrário de outras linguagens:
 - O *array* não tem de ser declarado com uma dimensão
 - Inclui **diferentes tipos de dados** no mesmo *array*
- Notação literal
 - Definidos com `[...]` e elementos separados por vírgulas

```
const values=[20,'array',5, true];  
console.log(values);
```



- Baseado num *constructor*:

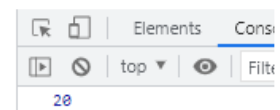
```
const valuesConstructor=new Array(20,'array',5,true)  
console.log(valuesConstructor)
```

Array

■ Indexação

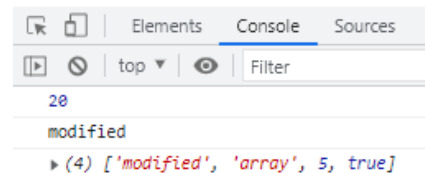
- nomeArray [*posição*]
- Índices iniciam-se em **zero**

```
const values=[20,'array',5, true];  
console.log(values[0]);
```



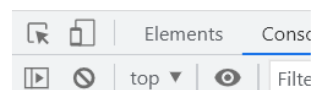
- Um array pode ser diretamente alterado

```
values[0]='modified';  
console.log(values[0]);  
console.log(values);
```



- Propriedade **length** é muito importante (retorna a dimensão do *array*)

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



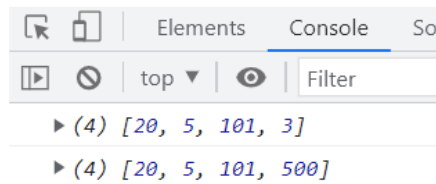
Array

- É usual declarar um array com base em const
 - Quando se define o array como const, na realidade define-se uma referência constante para o array o que não impede de alterar os seus elementos

```
const a=[20,5,101,3]
console.log(a)
a=[5,4]
console.log(a)
```

```
▶ (4) [20, 5, 101, 3]
✖ ▶ Uncaught TypeError: Assignment to constant variable.
```

```
const a=[20,5,101,3]
console.log(a)
a[3]=500
console.log(a)
```



```
Elements Console So
top Filter
▶ (4) [20, 5, 101, 3]
▶ (4) [20, 5, 101, 500]
```

- previne que a referência para o objeto não é alterada **o que não impede** a alteração das suas propriedades

Array

JavaScript Array Methods and Properties

| Name | Description |
|-------------------------------------|---|
| <u>concat()</u> | Joins arrays and returns an array with the joined arrays |
| <u>constructor</u> | Returns the function that created the Array object's prototype |
| <u>copyWithin()</u> | Copies array elements within the array, to and from specified positions |
| <u>entries()</u> | Returns a key/value pair Array Iteration Object |
| <u>every()</u> | Checks if every element in an array pass a test |
| <u>fill()</u> | Fill the elements in an array with a static value |
| <u>filter()</u> | Creates a new array with every element in an array that pass a test |
| <u>find()</u> | Returns the value of the first element in an array that pass a test |

...

...

https://www.w3schools.com/jsref/jsref_obj_array.asp

Exercício 2 (Array object)

Exercício 2

- Declare um array com os valores: mercedes, volvo, bmw, audi
- Determine qual o número de elementos do array (propriedade length)
- Ordene o array (método: sort())

■ Functions

- A forma literal é muito mais usada para a declaração de funções do que baseada num constructor, uma vez que é menos sujeita a erros e mais fácil de manter

```
function reflect(value){  
    return value;  
}
```



- Tendo por base um constructor seria:
 - Todo o corpo da função teria de ser encapsulado numa string, o que como é óbvio tem várias desvantagens (dificulta a implementação, dificulta o debug do código, ...)

```
var reflect=new Function("value","return value;");
```



- À excepção do *built-in type Function* (notação literal) não existe uma forma correta ou errada de instanciar *built-in types*.

Primitive Wrapper Types

Primitive Wrapper Types

- Existem três **Primitive Wrapper Types**:
 - *String*
 - *Number*
 - *Boolean*
- Possibilitam o funcionamento com os *Primitive Types* da mesma forma (***dot notation***) que ocorre com os *Reference Types*
- Ao contrário dos *reference type*, um *primitive wrapper type* **não permite a adição de propriedades**.

Strings

String Properties

| Property | Description |
|------------------------------------|---|
| <u>constructor</u> | Returns the string's constructor function |
| <u>length</u> | Returns the length of a string |
| <u>prototype</u> | Allows you to add properties and methods to an object |

String Methods

| Method | Description |
|---------------------------------------|---|
| <u>charAt()</u> | Returns the character at the specified index (position) |
| <u>charCodeAt()</u> | Returns the Unicode of the character at the specified index |
| <u>concat()</u> | Joins two or more strings, and returns a new joined strings |
| <u>endsWith()</u> | Checks whether a string ends with specified string/characters |
| <u>fromCharCode()</u> | Converts Unicode values to characters |
| <u>includes()</u> | Checks whether a string contains the specified string/characters |
| <u>indexOf()</u> | Returns the position of the first found occurrence of a specified value in a string |
| <u>lastIndexOf()</u> | Returns the position of the last found occurrence of a specified value in a string |

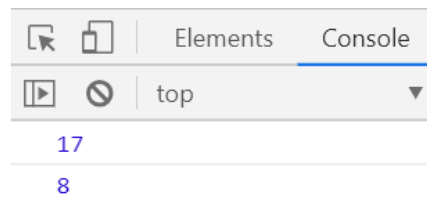
https://www.w3schools.com/jsref/jsref_obj_string.asp

Array

■ Propriedade (exemplo: length)

<script>

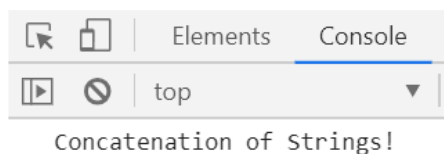
```
let firstStr='Concatenation of ';  
let secondStr='Strings!';  
console.log(firstStr.length);  
console.log(secondStr.length);
```



■ Método (exemplo: concat())

<script>

```
let firstStr='Concatenation of ';  
let secondStr='Strings!';  
console.log(firstStr.concat(secondStr));
```



Variáveis

| <u>Primitive Types</u> | <u>Reference Types (Built-in)</u> | <u>Primitive Wrapper Types</u> |
|------------------------|-----------------------------------|--------------------------------|
| Number | Object | Number |
| String | Array | String |
| boolean | Date | Boolean |
| null | Error | |
| undefined | Function | |
| | RegExp | |
| | Math | |
| | ... | |

Operadores

JS

Operadores

■ Aritméticos

| Operator | Description |
|----------|----------------------------------|
| + | Addition |
| - | Subtraction |
| * | Multiplication |
| ** | Exponentiation (<u>ES2016</u>) |
| / | Division |
| % | Modulus (Division Remainder) |
| ++ | Increment |
| -- | Decrement |

Operadores

■ Atribuição

| Operator | Example | Same As |
|----------|---------|------------|
| = | x = y | x = y |
| += | x += y | x = x + y |
| -= | x -= y | x = x - y |
| *= | x *= y | x = x * y |
| /= | x /= y | x = x / y |
| %= | x %= y | x = x % y |
| **= | x **= y | x = x ** y |

<http://www.w3schools.com/js>

Operadores

■ Lógicos

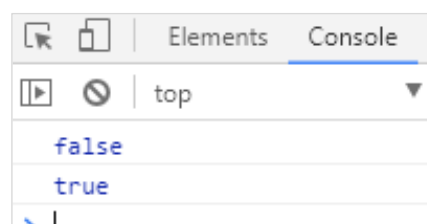
| Operator | Description |
|----------|-------------|
| && | logical and |
| | logical or |
| ! | logical not |

<http://www.w3schools.com/js>

```
<script>
  var a=5;
  var b=4;
  var c=6;
  var d=8;

  console.log((a>b)&&(c>d));

  console.log((a>b)||(c>d));
</script>
```



Operadores

■ Comparação

| Operator | Description |
|----------|-----------------------------------|
| == | equal to |
| === | equal value and equal type |
| != | not equal |
| !== | not equal value or not equal type |
| > | greater than |
| < | less than |
| >= | greater than or equal to |
| <= | less than or equal to |
| ? | ternary operator |

https://www.w3schools.com/js/js_comparisons.asp

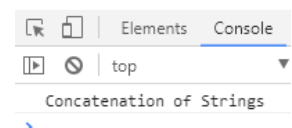
Operadores

■ Strings

- Concatenação (+)
 - Operador muito frequentemente utilizado

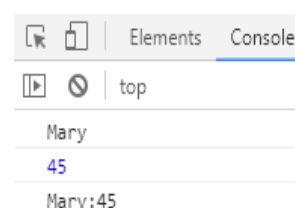
```
<script>  
  var strC = "Concatenation " + "of" + " Strings";  
  console.log(strC);  
</script>
```

Concatenação



- *type coercion* (um dos argumentos é uma *string*):

```
console.log(name);  
console.log(age);  
console.log(name + ":" + age);
```



Operadores

■ Precedência de Operadores

The following table is ordered from highest (20) to lowest (1) precedence.

| Precedence | Operator type | Associativity | Individual operators |
|------------|-----------------------------|---------------|----------------------|
| 20 | Grouping | n/a | (...) |
| 19 | Member Access | left-to-right | |
| | Computed Member Access | left-to-right | [...] |
| | new (with argument list) | n/a | new ... (...) |
| | Function Call | left-to-right | (...) |
| 18 | new (without argument list) | right-to-left | new ... |
| 17 | Postfix Increment | n/a | ... ++ |
| | Postfix Decrement | | ... -- |
| 16 | Logical NOT | right-to-left | ! ... |
| | Bitwise NOT | | ~ ... |
| | Unary Plus | | + ... |
| | Unary Negation | | - ... |
| | Prefix Increment | | ++ ... |
| | Prefix Decrement | | -- ... |
| | typeof | | typeof ... |
| | | | |

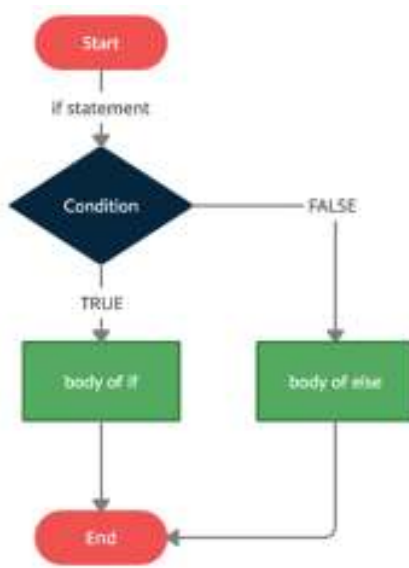
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence

Estruturas de Seleção (condicionais)

JS

Seleção

- if (condição) else ...



```
<script type="text/javascript">
    var teste = "verdadeiro";

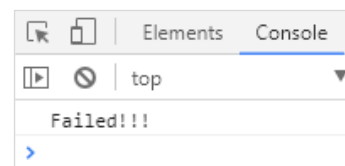
    if (teste == "verdadeiro")
        document.write("Condição Verdadeira!");
    else
        document.write("Condição Falsa!");
</script>
```

Seleção

- if ...else

```
<script>
    var threshold=50;
    var grade=40;

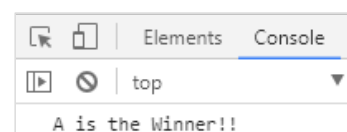
    if (grade>=threshold)
        {console.log("Aproved!!");}
    else
        {console.log("Failed!!");}
</script>
```



- condições encadeadas

```
<script>
    var scoreA=60;
    var scoreB=50;

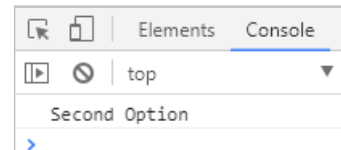
    if (scoreA>scoreB)
        {console.log("A is the Winner!!");}
    else if (scoreA<scoreB)
        {console.log("B is the Winner!!");}
    else
        {console.log("Draw !!")}
</script>
```



- switch (var)

```
{ case value1: statements; break;  
  case value2: statements; break; ...  
  default: statements }
```

```
<script>  
  var msg,a;  
  a=2;  
  
  switch(a)  
  {  
    case (1): msg="First Option"; break;  
    case (2): msg="Second Option"; break;  
    case (3): msg="Third Option"; break;  
    default: msg="No option!"; break;  
  }  
  console.log(msg);  
</script>
```



Estruturas de Repetição

JS

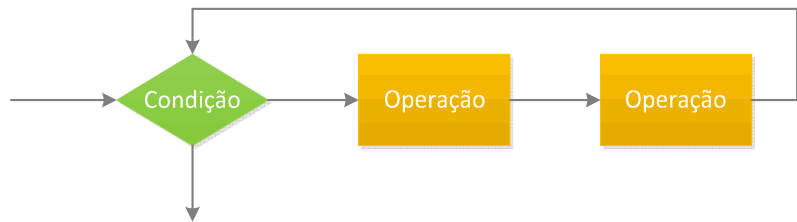
Repetição

■ Estruturas de Repetição / Ciclos (Loops)

■ **while (condição){**

// código bloco

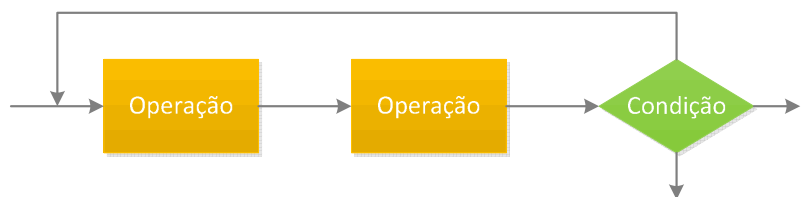
}



■ **do {**

// código bloco

} while (condição);



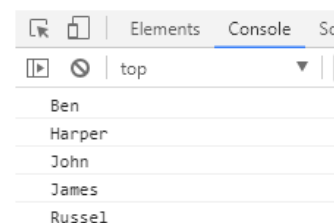
Repetição

■ Ciclos (Loops)

■ **for (initialization; condition; variable update) { ...}**

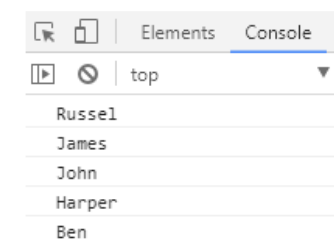
```
names=["Ben", "Harper","John", "James", "Russel"];

for(i=0; i<names.length; i++)
{
    console.log(names[i]);
}
```



```
names=["Ben", "Harper","John", "James", "Russel"];

for(i=names.length-1;i>=0;i--)
{
    console.log(names[i]);
}
```

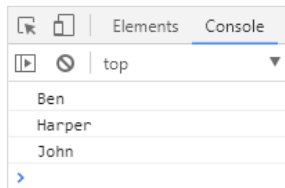


Repetição

■ Ciclos (Loops)

■ **break**

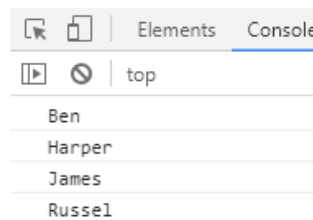
```
for(i=0; i<names.length; i++)
{
    console.log(names[i]);
    if(i===2)
        {break;}
}
```



break: interrompe o funcionamento do ciclo

■ **continue**

```
names=["Ben", "Harper","John", "James", "Russel"];
for(i=0; i<names.length; i++)
{
    if(i===2)
        {continue;}
    console.log(names[i]);
}
```



continue: salta diretamente para o final da iteração e prossegue com o ciclo, neste caso ignora a 3ª iteração

falsy / truthy

■ **falsy**

- valores tratados como **false**

| Value | Description |
|-----------------------------|------------------------------|
| var highScore = false; | false |
| var highScore = 0; | 0 |
| var highScore = ''; | empty value |
| var highScore = 10/'score'; | NaN (not a number) |
| var highScore; | variável sem valor atribuído |

■ **truthy**

- valores tratados como **true**

| Value | Description |
|-------------------------|-----------------------------|
| var highScore = true; | true |
| var highScore = 1; | número ≠ 0 |
| var highScore = 'xxxx'; | string com conteúdo |
| var highScore = 10/5 | resultado de um cálculo ≠ 0 |

Exercício 3 (Repetição)

Exercício 3

- Crie um array com a designação cars, contendo os valores:
 - 'mercedes','volvo','bmw','audi','kia','fiat','renault'
- crie um novo array (newcars) com base no array anterior, o qual contém apenas os valores correspondentes às posições pares
 - deve utilizar o método push()

Funções

JS

Funções

- Conjunto de declarações agrupadas para executar uma tarefa específica.
 - Reutilização de código; flexibilidade; ...
 - **Declaração de uma função** (notação literal):

```
function name (param1, param2, ....){  
    código a ser executado;  
}
```

```
function firstFunction(){  
    document.write("hello");  
}
```

- Prefixos uteis para nomes de função:
 - create, show, get, check,

Funções

- Chamada à função:
 - Efetuada através do nome da função seguido de parêntesis
 - Código só é executado após a respetiva chamada

```
firstFunction();
```

- O browser percorre todo o script antes da execução de cada declaração, mas preferencialmente a função deve ser declarada antes da sua chamada.

Funções

- Parâmetros
 - Declaração de uma função com parâmetros:

```
function calculateArea(width,height){  
    return width*height;  
}
```

- Chamada a uma função:
 - Especificação direta dos valores dos argumentos

```
calculateArea (2,4);
```

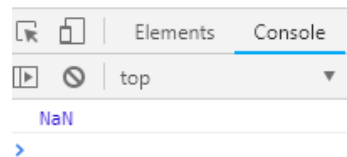
- Argumentos da função definidos através de variáveis

```
rectWidth=2;  
rectHeight=4;  
calculateArea(rectWidth, rectHeight);
```

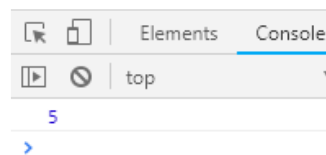
Funções

- Possível definir *default values* para os parâmetros

```
function calculateSum (a,b){  
    return a+b;  
};  
  
console.log(calculateSum(4));
```



```
function calculateSum (a,b = 1){  
    return a+b;  
};  
  
console.log(calculateSum(4));
```



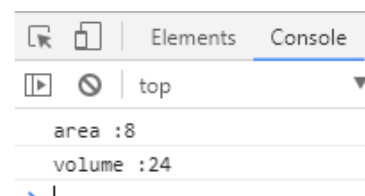
Funções

- Retorno de um valor único:

```
function calculateArea(width,height){  
    return width*height;  
}  
var wallOne=calculateArea(3,5);  
var wallTwo=calculateArea(8,5);
```

- Retorno de vários valores com base em *arrays*:

```
<script>  
  
    function getDimensions(w,h,d)  
    {  
        var area=w*h;  
        var volume=area*d;  
        var values=[area,volume];  
  
        return values;  
    }  
  
    console.log("area : " + getDimensions(2,4,3)[0]);  
  
    console.log("volume : " + getDimensions(2,4,3)[1]);  
  
</script>
```



Passagem de Parâmetros

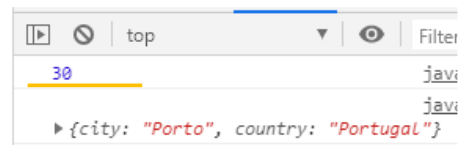
- Os *Primitive Type* e os Objetos são passados à função de forma diferente
 - Nos *Primitive Types* é feita a **passagem do valor** do argumento:
 - todas as alterações efetuadas no parâmetro no interior da função não alteram o valor original
 - Nos *Reference Types* a passagem é **feita por referência**:
 - as alterações feitas no interior da função são na realidade efetuadas no objeto original (objeto é referenciado pelas diversas variáveis)

```
<script>
  var age=30;
  var citizen={city:'Coimbra', country:'Portugal'};

  function changeValues(a,b){
    a=50;
    b.city='Porto'
  }

  changeValues(age,citizen);

  console.log(age);
  console.log(citizen);
</script>
```

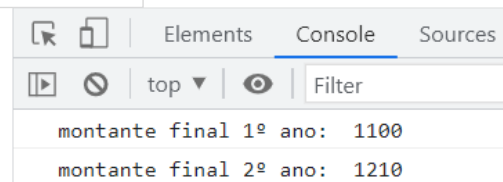


Exercício 4 (funções)

Exercicio 4

- Declare uma função que calcule a valorização de um depósito ao fim do primeiro e do segundo ano.
 - A função recebe o montante investido e a taxa de juro (fixa) e deve mostrar o montante ao fim do 1º e do 2º ano. (ex: montante 1000€, juros 10%)

```
function calculaJuros(m,j){  
  const values=[];  
  j/=100;  
  values[0]=m*(1+j);  
  values[1]=values[0]*(1+j);  
  return values  
}  
console.log('montante final 1º ano: ' + calculaJuros(1000,10)[0])  
console.log('montante final 2º ano: ' + calculaJuros(1000,10)[1])
```



Diferentes Tipos de Função

JS

Funções

- O JavaScript admite formas diferentes de criar uma função:

Declaração de Função

(statement)

```
function calculateSum (a,b = 1){  
    return a+b;  
}
```

Function Expression

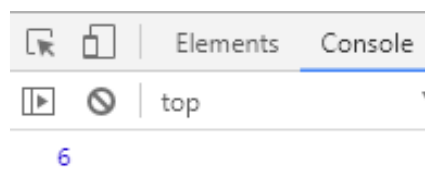
(anonymous function)

```
var calculateSum = function (a,b = 1){  
    return a+b;  
}
```

Funções

- Declaração de Função
 - Chamada à função:

```
<script>  
    var area;  
    area=calculaArea(2,3);  
  
    function calculaArea(width,height)  
    {  
        return width*height;}  
  
    console.log(area);  
</script>
```



A declaração normal de uma função permite que a chamada à função seja **executada antes** da declaração da função

Funções

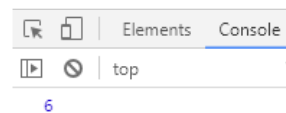
■ *Function Expression (anonymous function)*

- A declaração de uma função pode ser incorporada numa expressão
 - Não é especificado o nome da função depois de **function (anonymous function)**

```
<script>
  var calculaArea=function(width,height){
    return width*height;}

  var area=calculaArea(2,3);
  console.log(area);
</script>
```

A declaração de uma **anonymous function** exige que a chamada à função seja efetuada **depois da expressão**

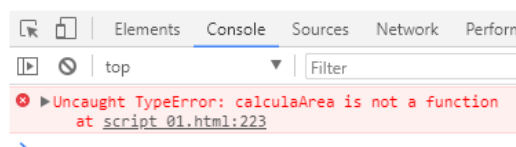


- É tratada como uma expressão, ou seja a função é detetada só após a sua chamada

```
<script>
  var area=calculaArea(2,3);
  var calculaArea=function(width,height){
    return width*height;}

  console.log(area);
</script>
```

Área não calculada, uma vez que a chamada à função foi feita antes da expressão onde está declarada



Funções

- A **anonymous function (function expression)** é particularmente importante no Javascript
 - definição de métodos de um objeto
 - event handling

```
var course={
  name:"web technologies",
  displayName:function(){
    document.write(course.name);
  }
}
course.displayName();
```

anonymous function define o método **displayName**

A definição de uma propriedade (nome:valor) é igual à definição de um método (nome:valor), neste último o valor é uma **anonymous function**



- O JavaScript admite formas diferentes de criar uma função:

Declaração de Função

```
function calculateSum (a,b = 1){  
    return a+b;  
}
```

- Sintaxe abreviada
- Palavra **function** é eliminada
- A seta => aponta para o corpo da função
- Caso não existam parâmetros são necessários parênteses vazios
- Se a função possuir várias declarações são necessárias chavetas e a key word **return**

Function Expression

(anonymous function)

```
var calculateSum = function (a,b = 1){  
    return a+b;  
}
```

Arrow Functions

```
var calculateSum = (a, b = 1) => a+b;
```