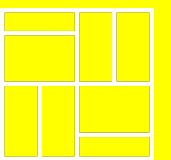


FUNÇÕES ANÓNIMAS

- Uma função anónima é uma função sem nome.
- Geralmente não está acessível após a sua criação.



Instanciar uma função

Uma função anónima não tem nome entre a palavra-chave `function` e os parênteses `()`:

```
var funcaoAnonima = function () {  
    console.log('isto é uma função anónima!');  
}  
  
funcaoAnonima();
```

Costumamos usar funções anónimas como argumentos de outras funções. Por exemplo:

```
setTimeout(function() {  
    console.log('isto é uma função dentro de um setTimeout!');  
}, 1000);
```

Neste exemplo, passamos por argumento uma função anónima para a função `setTimeout()`. A função passa a ser então um parâmetro da função `setTimeout`. A função `setTimeout()` executa essa função anónima um segundo depois.

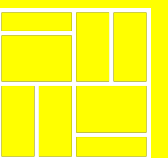
Exercício

FUNÇÕES ANÓNIMAS

- Criem uma função anónima que faça `console.log` do vosso nome completo.
- Criem uma função anónima que aceite 2 números e faça a sua soma.

IIFE

- Função anónima que é executada imediatamente após a sua declaração;
- IIFE = Immediately Invoked Function Expression;
- As variáveis definidas dentro da expressão não podem ser acedidas fora do seu contexto (scope), mesmo que seja um *var*.



Instanciar uma função

Uma função anónima não tem nome entre a palavra-chave `function` e os parênteses `()`:

```
(function (argumentos) {  
    console.log('Isto é uma IIFE');  
}) ('isto é um argumento');
```

Uma IIFE divide-se em duas secções:

FUNÇÃO

Nesta secção definimos a função anónima, que deverá estar dentro de parêntesis:

```
(function () {  
    console.log('Isto é uma IIFE');  
})
```

CHAMAR A FUNÇÃO

Após a definição da função adicionamos parêntesis de chamada de função, que podem ou não ter argumentos:

```
(argumentos);
```

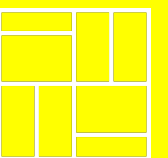
Exercício

IIFE

- Criem uma IIFE que faça `console.log` do vosso nome completo.
- Criem uma IIFE que aceite 2 números e faça a sua soma.

CLOSURES

- Funções definidas dentro de um "contexto léxico" (i.e. o corpo de uma função) acessam variáveis definidas nesse contexto;
- Uma closure dá acesso ao scope de uma função externa a partir de uma função interna;
- Closures são usadas para manter as variáveis privadas, não podendo ser acedidas externamente.



Definição

Consideremos o seguinte código:

```
function funcaoTeste() {  
    var mensagem = 'isto é uma mensagem!';  
  
    (function () {  
        console.log(mensagem);  
    })();  
}  
console.log( funcaoTeste() );    // 'isto é uma mensagem!'
```

A função *funcaoTeste()* é chamada e internamente executa a IIFE que está dentro do seu scope. Daí o resultado do `console.log` ser *'isto é uma mensagem!'*

Definição

Consideremos agora o seguinte código:

```
var primeiroValor = 5;

function funcaoTeste() {
    return function () {
        var segundoValor = 10;
        console.log(primeiroValor + segundoValor);
    };
}

var funcao = funcaoTeste();

console.log( funcao() );           // 15
console.log( primeiroValor );     // 10
console.log( segundoValor );      // ReferenceError: segundoValor is not defined
```

- A função *funcaoTeste()* ao ser chamada devolve uma função, e para ser executada essa função interna, deverá ser chamada novamente.
- A variável *segundoValor* não consegue ser acedida externamente.

Definição

Complicuemos agora um pouco:

```
function multiplicacao(primeiroValor) {  
    return function (segundoValor) {  
        return primeiroValor * segundoValor;  
    };  
}  
var funcao = multiplicacao(5);
```

```
console.log(multiplicacao(5));    // (segundoValor) { return primeiroValor * segundoValor; }  
console.log(funcao(2));           // 10  
console.log(multiplicacao(5)(2)); // 10
```

- O código `multiplicacao(5)` devolve a função interna, que é retornada pela função `multiplicacao`;
- O código `funcao(2)` devolve o valor correcto, porque `funcao` é o mesmo que `multiplicacao(5)`;
- O código `multiplicacao(5)(2)` devolve o valor correcto, porque ambas as funções são chamadas.

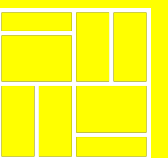
Exercício

CLOSURES

- Criem uma closure que divida por 2 um número que forneçam como parâmetro. Exemplo:
`funcao(5); // 2.5`
- Criem uma closure que faça uma divisão: o primeiro número é argumento da primeira função; o segundo argumento da divisão é passado para a função interna.

FUNÇÕES CONSTRUTORAS

- As funções construtoras em Javascript são como as classes do java (ou outras linguagens), diferenciando apenas pela sintaxe.
- Têm como propósito servir de molde para a criação de objetos.



Definição

Considerem o seguinte código de criação de uma função construtora:

```
function Carro(marca, modelo, ano){  
  this.marca = marca;  
  this.modelo = modelo;  
  this.ano = ano;  
}  
  
const carroNovo = new Carro('Nissan', 'Micra', '2018');  
  
console.log(carroNovo);
```

Para construir objetos, as funções construtoras precisam ser instanciadas pelo operador *new*. Por convenção, o seu nome deve conter a primeira letra maiúscula, mas não é obrigatório.

O *this* dentro delas é uma referência ao objeto criado a partir delas.

Leitura e alteração de valores

Como foi dito anteriormente, uma função construtora cria um Object, sendo a leitura e a alteração de valores similar. Consideremos o seguinte objecto:

```
function Carro(marca, modelo, ano){  
  this.marca = marca;  
  this.modelo = modelo;  
  this.ano = ano;  
}  
  
const carroNovo = new Carro('Nissan', 'Micra', '2018');
```

LEITURA

```
console.log(carroNovo.marca);    // Nissan
```

ALTERAÇÃO

```
carroNovo.marca = 'Renault';  
console.log(carroNovo.marca);    // 'Renault'
```

Exercício

FUNÇÕES CONSTRUTORAS

- Criem uma função construtora chamada Livro que aceite 2 argumentos: titulo e autor;
- Com a função que acabaram de construir, criem 2 exemplos de livros:
 - Os três mosqueteiros, Alexandre Dumas;
 - Os Maias, Eça de Queiroz.
- Façam `console.log` de todos os títulos e autores criados.
- Alterem o título do primeiro livro para: O Conde de Monte Cristo.