


## Trabalho 3 - DevWeb

### Closures - JavaScript

o que é?

Closures são objetos que se “lembram” do ambiente (escopo) em que foram criadas, normalmente criadas dentro de outras funções, tendo acesso às variáveis e os parâmetros que compõem a função pai da closure, mesmo que ela já tenha finalizado sua execução (CROCKFORD, 2001). Esse conceito é muito discutido pelos desenvolvedores, mesmo assim é considerado um conceito confuso para muitos.

A origem dessa confusão deriva que uma closure consiste de um tipo especial de objeto que combina duas coisas: a função e o ambiente onde a função foi criada, ou seja, ela incorpora tanto a função em que foi declarada quanto os objetos das variáveis que existiam no escopo em que foi declarada. Veja o exemplo a seguir:

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light green font and is numbered from 1 to 10 on the left side. It defines a function 'Exemplo()' which creates a local variable 'nome' with the value 'Adryan' and returns an inner function 'alertNome()' that calls 'alert(nome)'. Below this, the 'Exemplo()' function is called and stored in a variable 'myExemplo', which is then invoked.

```
1  function Exemplo() {  
2      var nome = "Adryan";  
3      function alertNome() {  
4          alert(nome);  
5      }  
6      return alertNome;  
7  }  
8  
9  var myExemplo = Exemplo();  
10 myExemplo();
```

A função `Exemplo()` cria uma variável local “nome” e declara uma função “`alertNome()`”, essa função é uma função aninhada, visto que ela está disponível apenas dentro da função `Exemplo()`, ademais ela não possui variáveis próprias, apenas utiliza a variável “nome” declarada externamente. Contudo, temos um “`return alertNome`” que retorna a closure antes mesmo dela ser executada fechando a função pai, deixando muitos confusos sobre como o código funciona sendo que normalmente uma variável local de uma função só funciona durante sua execução. Então como o “`alert(nome);`” ainda funciona?

A solução para esse problema está na função “myExemplo()”, visto que ela se torna na nossa closure, incorporando tanto a função “alertNome()” quanto as variáveis declaradas na função externa, por esse motivo o código funciona normalmente.

## IIFE - Immediately Invoked Function Expression:


Muitas vezes, no processo de desenvolvimento das funções nos arquivos .js, acabamos poluindo as variáveis e funções globais por declarar variáveis/funções locais com o mesmo nome resultando em conflitos entre as funções, mesmo sem percebermos. Para resolver esse problemas temos um dos padrões de design mais populares em JavaScript, a Expressão de Função Imediatamente Invocada (IIFE), a qual consiste em uma expressão de função que é executada imediatamente depois da sua declaração. Esse padrão gira em torno do uso do parêntese ( ), podendo ser dividido em duas partes: i) define uma função anônima cujo corpo é encapsulado entre parênteses, prevenindo que as variáveis locais entrem em conflito com as globais; ii) corresponde a criação da expressão ( ), expressa em seguida da declaração da função anônima, que irá executar imediatamente a função anônima. Veja os exemplos abaixo:

### 1 - Sem o uso de IIFE:



```
1 //exemplo1.js
2 function alertNome(nome) {
3     alert("Exemplo1: " + nome);
4 }
5
6 //Exemplo2.js
7 function alertNome(nome) {
8     alert("Exemplo2: " + nome);
9 }
```

Aqui criamos dois arquivos .js e declaramos cada função nesses arquivos, isto é, num arquivo terá uma função “alertNome(nome)” com o “alert(exemplo1 + nome)” e outro arquivo com a mesma função entretanto com o alert diferente. E inserimos os dois no documento HTML, como vemos abaixo, e no body criamos dois botões para chamar as funções tanto do exemplo1.js quanto do exemplo2.js, contudo só executa o alert do exemplo2.js, por causa do conflito com os nomes das funções, por esse motivo o navegador só interpretará a última definição.



```

1  <head>
2    <meta charset="UTF-8">
3    <meta http-equiv="X-UA-Compatible" content="IE=edge">
4    <meta name="viewport" content="width=device-width, initial-scale=1.0">
5    <title>IIFE</title>
6    <script src="./exemplo1.js"></script>
7    <script src="./exemplo2.js"></script>
8  </head>
9  <body>
10    <h1>Exemplo IIFE</h1>
11
12    <button onclick="alertName('Adryan')">Exemplo 1</button>
13    <button onclick="alertName('Cosmo')">Exemplo 2</button>
14  </body>

```

2 - Com o uso de IIFE:




```

1  var resultado1 = (function alertName() {
2    var nome = "Adryan";
3    function userName() {
4      alert("Exemplo1: " + nome);
5    };
6    return userName;
7  })();
8
9  //exemplo2.js
10 var resultado2 = (function alertName() {
11   var nome = "Cosmo";
12   function userName() {
13     alert("Exemplo2: " + nome);
14   };
15   return userName;
16 })();

```

Com o uso do IIFE podemos utilizar funções com o mesmo nome em arquivos diferentes ou até mesmo no mesmo arquivo sem preocupação, basta apenas que a variável que irá receber o IIFE tenha nomes diferentes, visto que agora, não temos mais duas funções com o mesmo

nome na hora de execução do JavaScript e sim duas variáveis que não armazenará a função em si, mas o resultado da função.



```
1 <head>
2   <meta charset="UTF-8">
3   <meta http-equiv="X-UA-Compatible" content="IE=edge">
4   <meta name="viewport" content="width=device-width, initial-scale=1.0">
5   <title>IIFE</title>
6   <script src="./exemplo1.js"></script>
7   <script src="./exemplo2.js"></script>
8 </head>
9 <body>
10  <h1>Exemplo IIFE</h1>
11
12  <button onclick="resultado1()">Exemplo 1</button>
13  <button onclick="resultado2()">Exemplo 2</button>
14 </body>
```