## Funções

As funções são objetos em JavaScript. Como as funções são objetos, elas se comportam de modo diferente das funções em outras linguagens, e entender esse comportamento é fundamental para ter uma boa compreensão do Javascript

## Declarações versus expressões

Há duas formas literais para as funções. A primeira é a declaração de função, que começa com a palavra-chave **function** e inclui o nome da função em seguida. O conteúdo da função é definido entre chaves, como mostrado no exemplo:

```
function add(num1, num2) {
    return num1 + num2;
}
```

A segunda forma é a expressão de função, que não exige um nome após a palavra-chave **function**. Essas funções são consideradas anônimas porque o objeto função propriamente dito não tem um nome. Em vez disso, as expressões de funções normalmente são referenciadas por meio de uma variável ou de uma propriedade, como no exemplo:

```
var add = function( num1, num2) {
    return num1 + num2;
};
```

Ambas as formas são quase idênticas, exceto pela ausência do nome da função e pelo ponto e vírgula no final da expressão função. Expressões de atribuição normalmente terminam com ponto e virgula, exatamente como ocorre na atribuição de qualquer outro valor

## Funções hoisted

Apesar de ambas as formas ( declaração de função e expressão) serem muito parecidas, elas diferem fundamentalmente em relação a um aspecto. As declarações de função são “içadas” ( hoisted) para o topo do contexto ( seja na função em que a declaração é feita, ou no escopo global) quando o código é executado. Isso significa que você pode definir uma função depois de ela ter sido utilizada no código, sem que um erro seja gerado. Observe o exemplo a seguir:

```
<!-- Com erro.-->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8"/>
    <script type="text/javascript">
     alert(add(1,2));
     var add = function( num1, num2) {
        return num1 + num2;
     }
    </script>
  </head>
```

```
  <body>
  </body>
</html>
```

```
<!-- SEM erro. -->

<html>
  <head>
    <meta charset="UTF-8"/>
    <script type="text/javascript">
     var add = function( num1, num2) {
        return num1 + num2;
     };
    alert(add(1,2));

    </script>
  </head>
  <body>
  </body>
</html>
```

```
<!-- SEM erro. -->

<html>
  <head>
    <meta charset="UTF-8"/>
    <script type="text/javascript">
     alert(add(1,2));

     function add ( num1, num2) {
        return num1 + num2;
     }
    </script>
  </head>
  <body>
  </body>
</html>
```

O hoisting de funções ocorre somente em declarações de funções porque o nome da função é previamente conhecido. Expressões de função, por outro lado, não podem sofrer hoisting porque as funções podem ser referenciadas somente por meio de uma variável.
Como o javascript tem funções de primeira classe, você pode usá-las assim como faria com qualquer objeto. Você pode atribuí-las a variáveis, adicioná-las a objetos, passá-las para outras funções como argumentos e retorná-las a partir de outras funções.
Considere o exemplo a seguir:

```
function sayHi() {
console.log("oi");
}
sayHi();

var sayHi2 = sayHi;
```

```
sayHi2();
```

Tanto sayHi quanto sayHi2 apontam para a mesma função, e isso significa que qualquer uma delas pode ser executada gerando o mesmo resultado.

```
var sayHi3 = new Function("console.log('Oi');");
sayHi3();

var sayHi4 = sayHi3;

sayHi4();
```

O construtor Funciton deixa mais explícito o fato de que sayHi3 possa ser atribuído como qualquer outro objeto. Quando você tem em mente que funções são objetos, muitos comportamentos começam a fazer sentido.

## Métodos de objetos

Você pode adicionar e remover propriedades dos objetos a qualquer momento. Quando um valor de uma propriedade é uma função, esse valor é considerado um método. Você pode adicionar um método a um objeto da mesma maneira que uma propriedade é adicionada.

```
var pessoa = {
   nome: “Orlando”,
   sayName: function() {
     console.log(pessoa.nome);
  }
};

pessoa.sayName; // exibe a função
pessoa.sayName();
```

Note que a sintaxe para o valor de uma propriedade de qualquer tipo e um método é a mesma: um identificador seguido de dois pontos e o valor. No caso, sayName, o valor é uma função.

O método sayName referencia pessoa.nome diretamente, o que gera um alto nível de acoplamento entre o método e o objeto. Isso representa um problema por vários motivos. M primeiro lugar, se o nome da variável for alterado, você terpa de se lembrar de alterar a referência a esse nome no método. Em segundo lugar, esse tipo de alto acoplamento faz com que seja difícil usar a mesma função em diferentes objetos. Felizmente, javascript há uma maneira de resolver esse problema.
Todo escopo em javascript tem um objeto this que representa o objeto que chama a função. No escopo global, this representa o objeto global (window, em web browsers). Quando uma função associada a um objeto é chamado, por padrão, o valor de this é igual a esse objeto. Portanto, em vez de referenciar diretamente um objeto em um método, this pode ser referenciado em seu lugar. Por exemplo:

```
var pessoa = {
   nome: “Orlando”,
   sayName: function() {
```

```
    console.log(this.nome);
  }
};

pessoa.sayName();
```

Com isso, você pode reutilizar uma função em objetos diferentes.

```
function meu_nome_eh() {
   console.log(this.nome);
}

var pessoa1 = {
    nome: “José”,
   sayName : meu_nome_eh
};

var pessoa2 = {
    nome: “Maria”,
   sayName : meu_nome_eh
};

var nome = “Orlando”;

pessoa1.sayName();
pessoa2.sayName();
meu_nome_eh();
```

## Definindo propriedades

Há duas formas de se criar seus próprios objetos em javascript: usando o construtor Object ou usnado um objeto literal.

```
var pessoa1 = {
   nome : “José”
};

var pessoa2 = new Object();

pessoa2.nome = “Maria”;

pessoa1.idade = 40;
pessoa2.idade = 30;
```

Como propriedades podem ser adicionadas a qualquer momento, às vezes é necessário verificar se uma propriedade existe no objeto. Desenvolvedores iniciantes utilizam padrões incorretos como o a seguir:

```
if ( pessoa1.idade) {
  // Faz algo com idade
}
```

O problema com esse padrão está no modo como as conversões de tipo do javascript aferam o resultado. A condição if é avaliada como true se o valor dor truthy( um objeto, uma string não vazia, um número diferente de zero ou true) e é avaliada como false se o valor for falsy ( null, undefined, 0, NaN ou string vazia). Como uma propriedade de objeto pode conter qualquer um desses valores falsy, o exemplo anterior pode resultar em um falso negativo.

O operador in procura a propriedade com um determinado nome em um objeto específico e retorna true se ela for encontrada.

```
console.log("nome" in pessoa1.idade);
console.log("idade" in pessoa1.idade);
```

## Removendo propriedades

Propriedades podem ser adicionadas e removidas a qualquer tempo:

```
var pessoa1 = {
   nome : "José",
   idade : 40
};

console.log("nome" in pessoa1); // true
console.log("idade" in pessoa1); // true

delete pessoa1.idade;

console.log("nome" in pessoa1);   // true
console.log("idade" in pessoa1);  // false
```

## Exercício

Crie uma solução para o problema conhecido como "FizzBuzz", em javascript