From *currying* to *closures* there are quite a number of JavaScript jargons (special words used within the field) knowing which will not only help you increase your vocabulary but understand JavaScript better. **Jargons are normally found in documentations and technical articles**. But some of them like *closures* are pretty standard things to know about. Knowing what the word itself mean can help you know the concept it is named for better.

This post is the compilation of 10 such terms with **their meaning** and **the context in which they are used** in JavaScript. If you're a beginner then this list has got you covered with the basics like *hoisting*. At the same time less-known or less-understood terms are also included in there.

1. Arity
2. Anonymous
3. Closure
4. Currying
5. Hoisting
6. Mutation
7. Pragma
8. Sentinel
9. Vanilla
10. Variadic

1. Arity

Arity (from Latin) is the term used to refer to the number of arguments or operands in a function

or operation respectively. You're most likely to come across this word in the realm of JavaScript when it is used to mention the **number of arguments expected by a JavaScript function**.

There is even a property named arity, of the `Function` object that returns the number of expected arguments in a function. It is now obsolete though and replaced by `length`.

The following function has an arity of 3.

```
function getName(first, middle, last){
    return first+' '+ middle +' '+last;
}
```

2. Anonymous

Anonymous is an adjective. When something or someone is referred to as anonymous it means that thing's or person's name is unidentified. Likewise in JavaScript an anonymous function is the one that is not identified by a name.

```
(function (){
  //body
})();
```

Above is an IIFE (Immediately Invoked Function Expression). The function in it is anonymous since it doesn't have a name. Now, take a look at the one below.

```
var foo = function() {

};
```

That is also said to be an anonymous function since there is no name after the key word `function`.

A little bit of doubt rises in the correctness of the use of the word "anonymous". With IIFE, the function gets called right away, no name involved whereas, to call the latter function the syntax `foo()` is used.

It's like we christened a nameless function with the name 'foo' and called it using that. Does that count as anonymous? I don't know, I'll leave that to the English experts. But, my confusion aside, they both are indeed referred to as anonymous function.

3. Closure

Here's one of the definitions from oxford dictionary for *closure*: "*A thing that closes or seals something, such as a cap or tie*."

In JavaScript, closure is an inner function, that is accessible outside of its outer function's scope, with its connection to the outer function's variables still intact.

To explain things (maybe not accurately but simply enough), consider *closure* as a waiter in a restaurant. A lot of things happen inside a restaurant kitchen, where we are not allowed to enter or see. But how are we supposed to get our food then?

That is where waiters come in. We call them, order the food, and then they'll go to the kitchen, inform the chefs of the orders, and bring it to us when the order is ready. This way we've not broken any "rules" and can still manage to grab a meal.

The waiter is someone who is able to take our order into the kitchen and return with the food. JavaScript *closures* are similar to that, they are able to **take our parameters** and **bring back us variables** (references to those variables, to be precise) from inside a function that we aren't allowed in.

```
function order() {
    var food;
    function waiter(order) {
        chef(order);
        return food;
    }
    function chef(order) {
        if (order === 'pasta') {
            food = ['pasta', 'gravy', 'seasoning'];
            cook();
        }
    }
    function cook() { food.push('cooked'); }
    return waiter;
}
var myOrder = order();
console.log(myOrder('pasta'));
// Array [ "pasta", "gravy", "seasoning", "cooked" ]
```

As you can see from the above code, everything apart from `waiter` and its return value from inside the order function isn't exposed to the outside world.

4. Currying

The effect, named after Haskell Curry, refers to **using multiple functions with single arguments**, in place of a single function with multiple arguments. Let's see the `add` functions below for example.

```
function addx(x){
    function addy(y){
        return x+y;
    }
    return addy
}

function add(x,y){
    return(x+y);
}

console.log(addx(3)(4)); \\7
console.log(add(3,4)); \\7
```

Both of the functions return the same result. The function `addx` accepts a parameter `x` while returning `addy` which in turn accepts the `y` value, performs the addition with `x` and returns the sum.

The function `add` simply takes both `x` and `y` at the same time, performs the addition and returns the sum. So far the first function might not seem very useful, until…

```
var add4 = addx(4);
console.log(add4(8)); //12
console.log(add4(6)); //10
console.log(add4(-74)); //-70
```

Now, the former function suddenly gets interesting. In currying, you can always fix a step in a sequence of operations like the addition of 4 from the above code, which is helpful when one of the variables used in the operation is always the same.

5. Hoisting

Hoist means to raise something. *Hoisting* in JavaScript also means the same and what gets raised is the declaration (variable & function declarations).

Declarations are where variables and functions are created with keywords `var`(not for global) and `function`.

It doesn’t matter where you type the code to declare a function or variable, during evaluation all the declarations are moved up inside the scope where they reside (except for in strict mode). Hence, it is possible to write a working code with the code for function call placed before function declaration.

```
var name = 'Velma';
console.log(sayCatchPhrase(name)); //"Jinkies!"

function sayCatchPhrase(name) {
    phrases = {
        'Fred Flintstone': 'Yabba dabba doo!',
        'Velma': 'Jinkies!',
        'Razor': 'Bingo!',
        'He-Man': 'I Have the Power'
    };
    return phrases[name];
}
```

[6. Mutation]

Mutation means change or modification. If you ever come across the word mutation in JavaScript it is probably referring to the changes that DOM elements went through.

There is even an API called MutationObserver to keep an eye out for the DOM mutations like **addition of child elements** or **changes to the element's attributes**. (You can read more about ~~MutationObserver~~ in my post.)

[7. Pragma]

Pragma is short for pragmatic information. In plain English, pragmatic is an adjective that means sensible and practical. In programming, *pragma* refers to the code that consist of useful information on **how a compiler or interpreter or assembler should process the program**.

It does not contribute anything to the programming language itself and its syntax may vary. They only affect the compiler behavior. JavaScript also has ~~few pragmas~~, one of them is `strict`.

```
"use strict";
```

By the above pragma, the JavaScript code will be executed in strict mode. In strict mode, bad syntax is not allowed, *hoisting* is not done, silent errors are shown, etc. It helps in **writing a more secure and optimized JavaScript code**.

[8. Sentinel]

Sentinels are soldiers who stand guard (Remember the ones from X-Men?). In programming, *sentinels* are values that are used to indicate the end of a loop or process. They can also be called “flags”.

You can use any reasonable value as a *sentinel*. Here's an example of *sentinels* used in JavaScript; the `indexOf` method which returns -1 (the sentinel value) when the search value is not found in the targeted string. Below is a function that returns the position of an array value and if value is not found, returns -1.

```
function getPos(ary, val) {
    var i=0, len=ary.length;
    for(;i<len;i++){
        if(ary[i]===val) return i+1;
    }
    return -1;
}
console.log(getPos(['r','y','w'],'y')); //2
console.log(getPos(['r','y','w'],'g')); //-1
```

9. Vanilla

I think everyone's first ice cream flavor must've been vanilla. I also think that not only in ice cream, but in pretty much every sweet dish vanilla kind of became *the*standard flavor. I've seen quite a few cake recipes where they add at least one drop of it into the mix – just to increase the flavor.

And that's what *vanilla* is, a **traditional standard flavor**. *Vanilla* JavaScript is referred to the standard JavaScript – no framework. Vanilla in fact is not only used to describe the standard version of JavaScript but also other languages like CSS.

10. Variadic

Variadic is an adjective created by joining "variable" and "adicity". "Adicity" is from ancient Greek, with a meaning that is the same as the Latin word "arity" (Item 1 in this list). Thus, the term *variadic* is used to **express something that has variable number of arguments**.

In JavaScript, a *variadic* function takes in any number of arguments. It can be created using `arguments` property, `apply` method and since ES6, the spread operator. Below is an example using a spread operator.

```
function test(...a){
  console.log(a);
}
test('a','b','c',8,[56,-89]);
```

```
//output is Array [ "a", "b", "c", 8, Array[2] ]

```