**JAVASCRIPT**

**Classes e function Prototypes:**

A diferença mais importante entre herança/inheritance baseada em class e em prototype é que a class define um tipo que pode ser instanciado em tempo de execução, enquanto prototype é a própria instância de objeto.

**Inheritance**

A herança/inheritcance em javascript se dá a partir de prototypes de objetos. portanto:

function Vehicle() {

this.Color = 'black';

this.start = function () {

console.log('Engine is on!');

}

}

let car = new Vehicle();

console.log(car.start()); //"Engine is on!"

console.log(Object.getPrototypeOf(car)); //Vehicle {}

O car instanciou um novo Vehicle, portanto herdou, em forma de prototype, os métodos e propriedades do Vehicle.

Quando *car.start()* for chamado, o JS irá verificar se o próprio car possui este método, se não tiver, irá verificar no prototype herdado, no caso Vehicle.  
Por sua vez, o Vehicle também herda um prototype, pois no javascript praticamente tudo herda de Object (exceto os primitives). Sendo assim, car.toString(), pode ser chamado, já que o prototype herdado de Vehicle, herdou de Object, que possui este método.

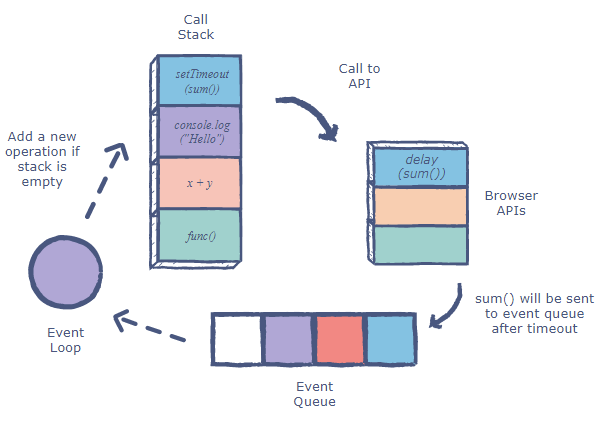
**\_\_proto\_\_ vs prototype**

\_\_proto\_\_ está depreciado, mas pode-se dizer que ele é um atalho de get e set para o prototype.

A diferença é que \_\_proto\_\_ é uma referência para a instância do objeto (o objeto pai) e prototype é referência para o construtor do objeto.

**Event Loop**

Javascript é uma linguagem single-thread e para operar dessa forma a engine utiliza o conceito de Event Loop. O Call Stack é responsável por definir a ordem que as operações devem ser executadas, a que está no topo será executada e removida da lista. O Event Queue é a fila de execução das funções.   
 Quando uma operação assíncrona é chamada, ela é enviada para o Browser API, que por sua vez vai aguardar o momento certo de executar e retornar a operação para a fila (por exemplo, um setTimeout). Assim, o Event Loop fica constantemente checando essas ordens, de forma infinita em loop. Se o call stack está livre, o event loop puxa operações do Event Queue para o call stack executar e se for operação assíncrona, é enviada para o Browser API que após execução volta para o fim da fila da Event Queue. Enquanto isso ocorre, o call stack segue sua execução, então um setTimeout sempre volta após toda execução.



**Event Loop: setTimeout() e Promise()**

Ambas são assíncronas e permitem o callstack seguir sua execução. Porém se um Promise for chamado depois de um setTimeout, e este resolver antes, ele terá prioridade.   
**setTimeout(fn, 0)**  
**new promise.resolve(‘done’)**  
ainda assim a promise retornará antes, pois promises geram uma “Microtask” no Browser API e essas tem prioridade sobre as “Macrotasks” que um setTimeout gera.

**Princípios do O.O:**

* **Encapsulation** (encapsulamento) -> Relativo a questões de segurança do código, como deixar variáveis protegidas ou públicas, ou somente sendo acessíveis por getters e/ou setters. É a ideia de não expor o código para fora do seu “escopo” ou classe.
* **Abstraction** (abstração) -> É referente a questão de “esconder” detalhes de implementação e exibir somente funcionalidade. Ajuda a reduzir duplicidade de código, criando implementações padronizadas para cada funcionalidade que será utilizada externamente.
* **Inheritance** (herança) -> É a reutilização de código. É o princípio de criar objetos, classes e controles que podem ser herdados para criar "filhos" semelhantes. Por exemplo: Um carro e uma moto são veículos, portanto se falarmos em termos de código, as classes carro e moto, herdam propriedades e funções da classe veículo, como por exemplo: ter motor, ter rodas, dar a partida, etc...
* **Polymorphism** (polimorfismo) -> Segue um conceito semelhante a inheritance, porém é quando uma classe que herda de outra, adapta uma função do seu “pai”, para se ajustar a sua necessidade. Por exemplo: uma classe “Eletrodoméstico” possui a função de Ligar(). Uma TV e uma geladeira herdam da classe Eletrodoméstico, porém, a função herdada “Ligar”, precisa ser adaptada para que cada um ligue da forma adequada a suas necessidades. Em muitas linguagens (não o JS), temos a palavra-chave Override, para indicar que um método será sobrescrita pelo herdeiro.

**Javascript types**

O JS trabalha com 7 tipos: string, number, bigInt, undefined, null, boolean e symbol. Há também o tipo composto: object. Os arrays são objetos, e seus enumeradores são numerados, ou seja, arrays nada mais são do que um atalho para um objeto: [“A”, “B”] é basicamente o mesmo que {0: “A”, 1: “B”}. Mas, é claro, os arrays também possuem funções específicas em seu protótipo, como o .sort() ou .reduce().

**null vs. undefined**

Apesar de não parecer ter diferença, são tipos diferentes. Uma variável sem valor atribuído, apenas instanciada, será undefined (*var a;*), porém é possível inicializar com valor nulo (*var a = null;*). Em testes a diferença pode ser notada com uso de igualdade estrita (===). Tal que:

null == undefined (true)  
null === undefined (false)

A diferença é semelhante a dizer que *undefined* é como se o elemento não existisse, e *null* é como se existisse, mas não possui valor algum.

**Pass-by-reference ou Pass-by-value:**

É sempre **pass-by-value**, mas para Objects o valor da variável é uma **referência**. Por isso, quando passa um Objeto e altera seus membros/propriedades, essas alterações persistem fora da função. Isso faz com que pareça passagem por referência. Mas se você realmente mudar o valor da variável do Objeto você verá que a mudança não persiste, provando que ela realmente passou por valor.

O confuso é que você não pode passar um Objeto, nem pode armazenar um Objeto em uma variável. Cada vez que você pensa que está fazendo isso, na verdade está passando ou armazenand6 Wouldn't it be starting at 16pm? uma **referência** a esse objeto. Mas quando você vai acessar seus membros/propriedades, ocorre uma “desreferenciação” silenciosa que perpetua a ficção de que sua variável continha o objeto real.

By-value:

let a = "1";

function change(param) {

param = "2";

}

change(a);

console.log(a); //"1"

Passing object reference:

let a = { value: "1" };

function change(param) {

param.value = 4;

}

change(a);

console.log(a); //{value: 4}

Passing object property (as value):

let a = { value: "1" };

function change(param) {

param = 4;

}

change(a.value);

console.log(a); //{value: "1"}

**Cookies vs. LocalStorage vs. SessionStorage**

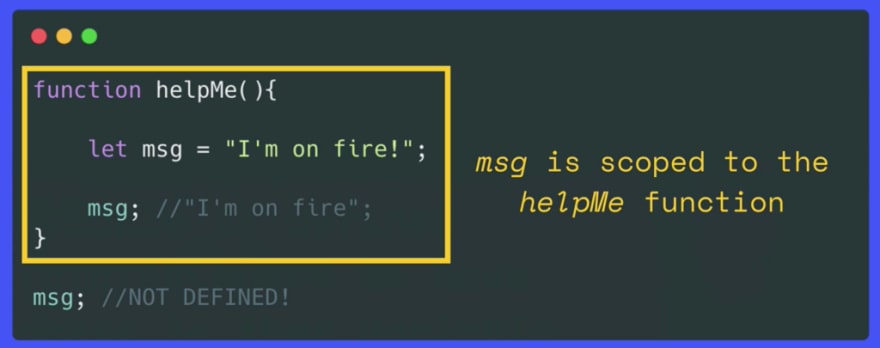
Cookies podem ser acessados pelo server-side e localStorage somente pelo client. Cookies são enviados juntos em requisições HTTP, num tamanho máximo de 4kb e possuem data de expiração. LocalStorage é armazenado no browser, com limite de 5mb. Já sessionStorage é igual ao local, porém os dados não persistem, se finalizar a sessão do browser, os dados salvos ali são apagados. Todos armazenam apenas strings.

**WebWorkers**

WebWorkers são uma forma de executar scripts JS em paralelo à thread principal. As funções executadas em um WebWorker não irão bloquear a execução de métodos da thread principal e serão executados de forma assíncrona e separada. Tão separada que a única forma de vínculo entre o código criador do Webworker e o próprio webWorker é por “postMessage”, ou seja é preciso enviar um informativo entre um e outro (em forma de string), solicitando execução de métodos ou recursos. Um WebWorker não tem acesso ao DOM e várias propriedades e métodos da Window também não estão disponíveis. Os dados transferidos entre thread e webWorker são copiados e não compartilhados. Pode ser um Dedicated ou Shared. Dedicated serve a um script enquando shared é compartilhado por vários scripts, iFrames e windows, desde que esteja dentro do mesmo domínio/host.

**Escopos**

Function -> Escopo interno de uma função, variáveis criadas aqui não são acessíveis de fora



Block -> Os blocos de código são definidos com chaves. É importante observar que *let* e ​​*const* têm escopo de bloco, já *var* não, devido ao hoisting (ver mais abaixo)

//let or const ARE BLOCK SCOPED

if (true) { //<---- block scope

const animal = "bird";

} //---> end of block scope

console.log(animal); //NOT DEFINED!

//var is NOT BLOCK SCOPED

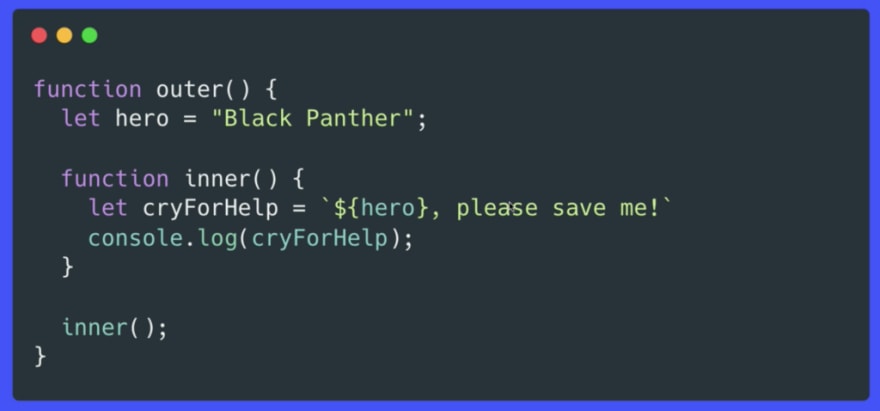
if (true) {

var animal = "bird";

}

console.log(animal); //'bird'

Lexical -> O escopo léxico se refere ao fato de que as funções aninhadas têm como escopo as funções pai/externas (mas é um relacionamento unilateral). No exemplo a seguir, a função interna está aninhada dentro da função externa. Como pode ver, temos acesso a variável ‘hero’ dentro da função interna. Mas fora dessa função não temos acesso ao interno.



**VARIABLE HOISTING (var)**

//this is variable HOIST

function hey() {

a = "I exist";

}

hey();

console.log(a); //it will log "I exist"

var a; //declared just here

**Logaritmos**

Log2(8) = 2?(8) -> 8/2 = **4**/2 = **2**/2 = **1 -> (23=8)**   
(foram feitas 3 divisões pela base até resultar 1. Normalmente, ciências da computação usam log base 2 -> log2)

Log10(100) = 10?(100) -> 100/10 = **10**/10 = **1 -> (102=100)**   
(foram feitas 2 divisões pela base até resultar 1. Logaritmos padrões usam log base 10 -> log10)