# Javascript

## ¿Qué es?

Es un lenguaje de scripting orientado a objetos multiplataforma, que requiere de un entorno de ejecución anfitrión para ejecutar (Un navegador, Node JS). El lenguaje se encuentra estandarizado por ECMAScript (ES). Los siguientes motores de ejecución que siguen el estándar ChakraCore, V8, SpiderMonkey entre otros.

## Declaración de variables

- const: Permite declarar e inicializar constantes de solo lectura
- var: Permite declarar variables
- let: Permite declarar variables con alcance de bloque

## Primitivas

- boolean: true o false.
- number: Números en general enteros o de punto flotante; pero se debe tener en cuenta que Javascript siempre almacena los números como de punto flotante, siguiendo el estándar IEEE 754.

  | Mantisa | Exponente | Signo |
  |---|---|---|
  | 52 bits (0 - 51) | 11 bits (52 - 62) | 1 bit (63) |

  De lo anterior podemos concluir que por defecto en Javascript podemos representar con precisión números enteros hasta de 15 dígitos y de punto flotante hasta 17 posiciones decimales; pero para estos últimos no hay precisión garantizada.

  Existen dos propiedades de nivel superior de tipo numérico que son:

  - *Infinity:* sirve para representar cálculos numéricos que exceden la capacidad de los números en Javascript.
  - *NaN:* se usa para indicar que un valor no es un número.

- undefined: Propiedad de nivel superior que indica que una variable no ha sido inicializada.
- string: Sirve para almacenar cadenas de texto.

- symbol: Tipo de dato cuyas instancias son únicas e inmutables.

- null: Palabra reservada usada para denotar un valor nulo.

## Estructuras de control

- Bloque: Es usada para agrupar instrucciones y se delimita usando llaves.

- If else: Funciona igual que el condicional en Java con la diferencia que los siguientes valores pueden ser evaluados como falsos:
    - undefined
    - null
    - 0
    - NaN
    - '' o ""
- switch: Funciona igual que en Java.

- ciclos: En Javascript los ciclos funcionan igual que en Java; pero existen los siguientes ciclos adicionales:

    - for … in: Itera a lo largo las propiedades contenidas en un objeto o arreglo.
    - for … of: Itera a lo largo de los valores contenidos en un objeto iterable (arreglos, sets, maps).

## Declarando funciones:

En Javascript las funciones pueden ser funciones nombradas o anónimas siguiendo las siguientes sintaxis:

- Nombradas:

```
function miFuncion(argumento1, argumento2) {
     …
}
```

- Anónimas:

```
function(argumento1, argumento2) {
     …
}
```

Las funciones en Javascript pueden ser asignadas a variables y posteriormente ser llamadas a través de dicha variable:

```
var iterar = function(argumento1) {
  for(let item of argumento1){
    console.log(item);
  }
};
iterar([1, 2, 3]);
iterar.call(this, [1, 2, 3]);
iterar.apply(this, [[1, 2, 3]]);
```

## Scope chaining (Closures)

En Javascript las funciones pueden ser anidadas; lo que permite a las internas acceder al alcance o contexto de las funciones que las contienen; pero las funciones externas no tienen acceso al alcance o contexto de las funciones anidadas.

Ejemplo:

```
function velocidad(velocidadInicial){
  this.velocidad = velocidadInicial;
  return function(aceleracion){
    this.velocidad += aceleracion;
    return this.velocidad;
  }
}
var acelerar = velocidad(0);
console.log(acelerar(10));
```

## Error común

```
function showMessage(message) {
  console.log(message);
}

function setupMessages() {
  var messageText = [
      'Message 1',
      'Message 2',
      'Message 3'
    ];
  for(var message of messageText){
    setTimeout(function(){showMessage(message)});
  }
}
setupMessages();
```

*Solución previa a ES6*

```
function showMessage(message) {
  console.log(message);
}

function makeCallback(message){
  return function(){
    showMessage(message);
  }
}

function setupMessages() {
  var messageText = [
      'Message 1',
      'Message 2',
      'Message 3'
    ];
  for(var message of messageText){
    setTimeout(makeCallback(message));
  }
}
setupMessages();
```

*Solución ES6*

```
function showMessage(message) {
  console.log(message);
}

function setupMessages() {
  var messageText = [
      'Message 1',
      'Message 2',
      'Message 3'
    ];
  for(let message of messageText){
    setTimeout(function(){showMessage(message)});
  }
}
setupMessages();
```

## Arrow functions

Son funciones anónimas usadas para simplificar y reemplazar algunas function expressions, y poseen las siguientes características.

- No crean su propio contexto (this), encapsulan el contexto que la envuelve.
- La propiedad arguments de la función hace referencia a los argumentos pasados al alcance que la envuelve.

Ejemplos:

```
function showMessage(message) {
  console.log(message);
}

function setupMessages() {
  var messageText = [
      'Message 1',
      'Message 2',
      'Message 3'
    ];
  for(let message of messageText){
    setTimeout(() => showMessage(message));
  }
}
setupMessages();



var unoADiez = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
var sumatoria = unoADiez.reduce(function(a, b){return a +
b;});
var sum = unoADiez.reduce((a, b) => a + b);
console.log(`${sumatoria} ${sum}`);
var impares = unoADiez.filter(function(v){return v % 2 ==
1;});
var odd = unoADiez.filter(v => v % 2 == 1);
console.log(`${impares} ${odd}`);
var personas = [
  {
    name: "Luis",
    surname: "Sanchez"
  },
  {
```

```
    name: "Pedro",
    surname: "Picapiedra"
  },
  {
    name: "Fusajiro",
    surname: "Yamauchi"
  },
  {
    name: "Bill",
    surname: "Gates"
  }
];
var nombres = personas.map(function(persona){return
persona.name;});
var nombresArrowReturn = personas.map(persona => {return
persona.name;});
var names = personas.map(persona => persona.name);
console.log(`${nombres} ${nombresArrowReturn} ${names}`);
```

## Palabra reservada arguments

Sirve para acceder al arreglo de argumentos recibidos por una función.

## Parámetros por defecto

En Javascript podemos asignar valores por defecto a los parámetros definidos por una función, colocando un igual seguido por el valor que queremos por defecto en caso que en la instrucción que llama la función, no sea pasado el o los argumentos. Los argumentos por defecto están disponibles para usarse en parámetros por defecto posteriores.
Las funciones usadas como parámetros por defecto no pueden ser accedidas por las funciones anidadas.

Ejemplos:

```
function parametrosPorDefecto(x=10){
  console.log(x);
}
function parametrosPorDefectoConfuncion(x=getDefault()){
  console.log(x);
}
function getDefault(){
  return 50;
```

```
}
function parametrosObligatorios(x=validateRequire('x')){
  console.log(x);
}
function validateRequire(message){
  throw new Error(`El parámetro ${message} es obligatorio`);
}
parametrosPorDefecto();
parametrosPorDefectoConfuncion();
parametrosObligatorios();
```

## Herencia

Debido a que en Javascript no tenemos clases, solo instancias, usamos la propiedad prototype de cada objeto para definir y modificar sus métodos.

Ejemplo:

### Previa a ES6

```
function Vehiculo(medios){
  this.medios = medios;
  this.aceleracion = 0;
  this.velocidad = 0;
}
Vehiculo.prototype.acelerar = function(metrosSegundos2){
  this.aceleracion += metrosSegundos2;
}
Vehiculo.prototype.getMedios = function(metrosSegundos2){
  return this.medios;
}
Vehiculo.prototype.getAceleracion = function(){
  return this.aceleracion;
}
Vehiculo.prototype.getVelocidad = function(){
  return this.velocidad;
}
Vehiculo.prototype.toString = function(){
  return this.medios.join(' ') + ' ' + this.aceleracion + ' '
+ this.velocidad;
}
function Bicicleta(){
  Vehiculo.call(this, ['terrestre']);
}
Bicicleta.prototype = Object.create(Vehiculo.prototype);
var aceleracionHeredada = Bicicleta.prototype.acelerar;
Vehiculo.prototype.acelerar = function(metrosSegundos2){
  console.log('Pedaleando');
  aceleracionHeredada.call(this, metrosSegundos2);
}
var giant = new Bicicleta();
giant.acelerar(15);
console.log(giant.toString());
```

## Usando self-invoking functions para definir los métodos

```
function Vehiculo(medios){
  this.medios = medios;
  this.aceleracion = 0;
  this.velocidad = 0;
}
(function() {
    this.acelerar = function(metrosSegundos2){
      this.aceleracion += metrosSegundos2;
    };
    this.getMedios = function(metrosSegundos2){
      return this.medios;
    };
    this.getAceleracion = function(){
      return this.aceleracion;
    };
    this.getVelocidad = function(){
      return this.velocidad;
    };
    this.toString = function(){
      return this.medios.join(' ') + ' ' + this.aceleracion +
' ' + this.velocidad;
    };
}).call(Vehiculo.prototype);
function Bicicleta(){
  Vehiculo.call(this, ['terrestre']);
}
Bicicleta.prototype = Object.create(Vehiculo.prototype);
var aceleracionHeredada = Bicicleta.prototype.acelerar;
Vehiculo.prototype.acelerar = function(metrosSegundos2){
  console.log('Pedaleando');
  aceleracionHeredada.call(this, metrosSegundos2);
}
var giant = new Bicicleta();
giant.acelerar(15);
console.log(giant.toString());
```

## Usando ES6

```
class Vehiculo {

  constructor (medios){
    this._medios = medios;
    this._aceleracion = 0;
    this._velocidad = 0;
  }

  acelerar(metrosSegundos2) {
    this._aceleracion += metrosSegundos2;
  }

  get medios(){
    return this._medios;
  }

  set medios(medios){
    this._medios = medios;
  }

  get aceleracion(){
    return this._aceleracion;
  }

  set aceleracion(aceleracion){
    this._aceleracion = aceleracion;
  }

  get velocidad(){
    return this._velocidad;
  }

  set velocidad(velocidad){
    this._velocidad = velocidad;
  }

  toString(){
    return this.medios.join(' ') + ' ' + this.aceleracion + '
' + this.velocidad;
  }
}

class Bicicleta extends Vehiculo {
```

```
  constructor (){
    super(['terrestre']);
  }

  acelerar(metrosSegundos2) {
    console.log('Pedaleando');
    super.acelerar(metrosSegundos2);
  }
}
var giant = new Bicicleta();
giant.acelerar(15);
console.log(giant.toString());
```

# Patrones de diseño

## ¿Qué es un patrón de diseño?

Es una solución reusable que puede ser aplicada a problemas que ocurren con frecuencia en el diseño de software. Los patrones de diseño generales descritos por GoF pueden categorizarse en los siguientes tres grupos:

- *Creacionales:*

Se enfocan en los mecanismo de creación de los objetos en los cuales los objetos son creado de una manera adecuada para el problema en que se está trabajando.

- *Estructurales:*

Se encargan de definir la composición de los objetos y típicamente identifican formas simples de entender las relaciones entre diferentes objetos.

- *Comportamentales:*

Se encargan de mejorar y racionalizar la comunicación entre objetos diferentes en un sistema.

Para mayor información:

http://www.oodesign.com/

## Reveling Module

Permite definir propiedades y métodos privados de un objeto, blindándolo del contexto global.

Ejemplo:

```
var logger = (function(salida){
  var levels =  {
    DEBUG: {"value": 1},
    INFO: {"value": 2},
    WARNING: {"value": 3},
    ERROR: {"value": 4}
  };
  var exportedLogger = {};
  var maxLogLevel = levels.DEBUG;
  function log(level, message){
    switch(level){
      case levels.DEBUG: {
        if(maxLogLevel.value <= 1){
          salida.debug("DEBUG: %s", message);
        }
        break;
      }
      case levels.INFO: {
        if(maxLogLevel.value <= 2){
          salida.info("INFO: %s", message);
        }
        break;
      }
      case levels.WARNING: {
        if(maxLogLevel.value <= 3){
          salida.warn("WARNING: %s", message);
        }
        break;
      }
      case levels.ERROR: {
        if(maxLogLevel.value <= 4){
          salida.error("ERROR: %s", message);
        }
        break;
      }
    }
  }
```

```
  function setLogLevel(level){
    maxLogLevel = level;
  }

  return {
    level: levels,
    log: log,
    setLogLevel: setLogLevel
  };
})(console);
logger.setLogLevel(logger.level.DEBUG);
logger.log(logger.level.DEBUG, "Hola mundo");
logger.log(logger.level.INFO, "Hola mundo");
logger.log(logger.level.WARNING, "Hola mundo");
logger.log(logger.level.ERROR, "Hola mundo");
```

## Singleton

Es usado para restringir la instanciación de una clase a un solo objeto (una única instancia).

Ejemplo:

```
var configuration = (function () {

  var instance;

  function init() {

    var properties = {};

    return {

      getProperty: function (propertyKey) {
        return properties[propertyKey];
      },

      setProperty: function(propertyKey, propertyValue){
        properties[propertyKey] = propertyValue;
      }
    };

  };

  return {

    getInstance: function () {

      if ( !instance ) {
        instance = init();
      }

      return instance;
    }

  };

})();

var firstReference = configuration.getInstance();
```

```
firstReference.setProperty('welcomeMessage', 'Hello world');
var secondReference = configuration.getInstance();
console.log(secondReference.getProperty('welcomeMessage'));
```

## Observer

Es usado cuando es necesario que uno o más objetos a los que denominamos observers (observadores) sean notificados por los cambios de estado de un objeto al que llamamos subject (tema o sujeto de observación).

Ejemplo:

```
function ObserverList(){
  this.observerList = [];
}
(function(){
  this.add = function( obj ){
    return this.observerList.push( obj );
  };

  this.count = function(){
    return this.observerList.length;
  };

  this.get = function( index ){
    if( index > -1 && index < this.observerList.length ){
      return this.observerList[ index ];
    }
  };

  this.indexOf = function( obj, startIndex ){
    var i = startIndex;
    while( i < this.observerList.length ){
      if( this.observerList[i] === obj ){
        return i;
      }
      i++;
    }
    return -1;
  };

  this.removeAt = function( index ){
```

```
    this.observerList.splice( index, 1 );
  };
}).call(ObserverList.prototype);


function Subject(){
  this.observers = new ObserverList();
}
(function(){
  this.addObserver = function( observer ){
    this.observers.add( observer );
  };

  this.removeObserver = function( observer ){

this.observers.removeAt( this.observers.indexOf( observer,
0 ) );
  };

  this.notify = function( context ){
    var observerCount = this.observers.count();
    for(var i=0; i < observerCount; i++){
      this.observers.get(i).update( context );
    }
  };
}).call(Subject.prototype);

function extend( obj, extension ){
  for ( var key in extension ){
    obj[key] = extension[key];
  }
}

function Observer(){
  this.update = function(){
  };
}

window.onload = function () {
  var buttons = document.getElementsByTagName('button');
  function requestStarts(event){
    let button = event.target;
    button.notify('start');
    setTimeout(function(){
      button.notify('end');
    }, 5000);
```

```
  }
  function requestEnds(value){
    switch(value){
        case 'start': {
          this.disabled = true;
          break;
        }
        default:{
          this.disabled = false;
        }
      }
  }
  for(let button of buttons){
    extend( button, new Subject() );
    button.onclick = requestStarts;
    extend( button, new Observer());
    button.update = requestEnds;
    for(let innerReference of buttons){
      button.addObserver( innerReference );
    }
  }
};
```

## Command

Encapsula los métodos permitiendo realizar llamados a estos sin la necesidad de conocer el método o el objeto sobre el que se ejecuta.

Ejemplo:

```
var calculadora = (function(){
  var _self = this;
  _self.sumar = function(a, b){
    return a + b;
  }
  _self.restar = function(a,b){
    return a - b;
  }
  _self.multiplicar = function(a,b){
    let result = 0;
    for(let i = 0; i < a;i++){
      result = _self.sumar(result, b);
    }
```

```
    return result;
  }
  function execute(commandName) {
    return _self[commandName] &&
_self[commandName].apply( _self, [].slice.call(arguments,
1) );
  }
  return {
    execute: execute
  };
})();
console.log(calculadora.execute('sumar', 10, 23));
console.log(calculadora.execute('multiplicar', 10, 23));
```

## Factory

Permite crear objetos sin exponer al cliente el proceso de inicialización; se usa cuando el proceso de creación de los objetos presenta un alto grado de complejidad o poseemos múltiples clases que comparten las mismas propiedades.

Ejemplo:

```
function NonBlockingMessage(message){
  function showMessage(){
    let messageContainer =
document.getElementById('messageContainer');
    let messageHolder = document.createElement('div');
    messageHolder.innerHTML = message;
    messageContainer.appendChild(messageHolder);
  }
  return {
    showMessage: showMessage
  }
}
function BlockingMessage(message){
  function showMessage(){
    alert(message);
  }
  return {
    showMessage: showMessage
  }
}
function MessageHandlerFactory(){
```

```
}
(function(){
  this.create = function(type, message){
    switch(type){
      case 'nonBlocking':{
        return new NonBlockingMessage(message);
      }
      default:{
        return BlockingMessage(message);
      }
    }
  }
}).call(MessageHandlerFactory.prototype);
var messageFactory = new MessageHandlerFactory();
function showBlockingMessage(message){
  messageFactory.create('blocking', message).showMessage();
}
function showNonBlockingMessage(message){
  messageFactory.create('nonBlocking',
message).showMessage();
}
```

## Decorator

Permite extender la funcionalidad de un objeto en tiempo de ejecución (agregar nuevas responsabilidades a un objeto en tiempo de ejecución).

Ejemplo:

```
function NonBlockingMessage(message){
  function showMessage(){
    let messageContainer =
document.getElementById('messageContainer');
    let messageHolder = document.createElement('div');
    messageHolder.innerHTML = message;
    messageContainer.appendChild(messageHolder);
  }
  return {
    showMessage: showMessage
  }
}
function BlockingMessage(message){
  function showMessage(){
    alert(message);
  }
  return {
    showMessage: showMessage
  }
}
function MessageHandlerFactory(){
}
(function(){
  this.create = function(type, message){
    switch(type){
      case 'nonBlocking':{
        return new NonBlockingMessage(message);
      }
      default:{
        return BlockingMessage(message);
      }
    }
  }
}).call(MessageHandlerFactory.prototype);
var messageFactory = new MessageHandlerFactory();
function showBlockingMessage(message){
  messageFactory.create('blocking', message).showMessage();
}
```

```
function showNonBlockingMessage(message){
  messageFactory.create('nonBlocking',
message).showMessage();
}
function OnlyOneMessageDecorator(wrappedInstance){
  function clearAndShow(){
    let messageContainer =
document.getElementById('messageContainer');
    messageContainer.innerHTML = '';
    wrappedInstance.showMessage();
  }
  return {
    showMessage: clearAndShow
  }
}
function showOnlyOneMessage(message){
  var decoratedMessage = new
OnlyOneMessageDecorator(messageFactory.create('nonBlocking',
message));
  decoratedMessage.showMessage();
}
```