JAVASCRIPT FRONTEND AVANZADO

CLASE 8 : PROTOTIPOS

PROTOTIPOS

JavaScript provoca cierta confusión en desarrolladores con experiencia en lenguajes basados en clases (como Java o C++), por ser dinámico y no proporcionar una implementación de clases en sí mismo (la palabra clave class se introdujo en ES2015, pero sólo para endulzar la sintaxis, ya que JavaScript sigue estando basado en prototipos).

En lo que a herencia se refiere, JavaScript sólo tiene una estructura: **objetos**. Cada objeto tiene una propiedad privada (referida como su [[Prototype]]) que mantiene un enlace a otro objeto llamado su prototipo. Ese objeto prototipo tiene su propio prototipo, y así sucesivamente hasta que se alcanza un objeto cuyo prototipo es null. Por definición, null no tiene prototipo, y actúa como el enlace final de esta cadena de prototipos.

https://hackernoon.com/prototypes-in-javascript-5bba2990e04b

Casi todos los objetos en JavaScript son instancias de Object que se sitúa a la cabeza de la cadena de prototipos.

A pesar de que a menudo esto se considera como una de las principales debilidades de JavaScript, el modelo de herencia de prototipos es de hecho más potente que el modelo clásico. Por ejemplo, es bastante simple construir un modelo clásico a partir de un modelo de prototipos [1].

Los lenguajes orientados a objetos basados en clases, como Java y C++, se basan en la existencia de dos entidades distintas: clases y ejemplificaciones de estas -mal llamadas instancias-.

- Una *clase* define todas las propiedades (considerando como propiedades los métodos y campos de Java, o los miembros de C++) que caracterizan un determinado conjunto de objetos. Una clase es una entidad abstracta, en lugar de algún miembro en particular del conjunto de objetos que describe. Por ejemplo, la clase Employee puede representar el conjunto de todos los empleados.
- Un *ejemplo (o "una instancia")*, por otro lado, es la ejemplificación de una clase; es decir, uno de sus miembros. Por ejemplo, Victoria podría ser una ejemplificación de la clase Employee, representando a un individuo en particular como un empleado. Una ejemplificación de una clase ("instance") tiene exactamente las propiedades de su clase padre (ni más, ni menos).

Un lenguaje basado en prototipos, como JavaScript, no hace esta distinción: simplemente tiene objetos. Un lenguaje basado en prototipos tiene la noción del *objeto prototípico*, un objeto que se utiliza como una plantilla a partir de la cual se obtiene el conjunto inicial de propiedades de un nuevo objeto. Cualquier objeto puede especificar sus propias propiedades, ya sea cuando es creado o en tiempo de ejecución. Adicionalmente, cualquier objeto puede ser utilizado como el *prototipo* de otro objeto, permitiendo al segundo objeto compartir las propiedades del primero [2].

Cada objeto en JavaScript tiene un segundo objeto JavaScript (o nulo, pero esto es raro) asociado a él. Este segundo objeto se conoce como prototipo y el primer objeto hereda propiedades del prototipo. Todos los objetos creados por literales de objeto tienen el mismo objeto prototipo, y podemos referirnos a este objeto prototipo en código JavaScript como Object.prototype, pero recordemos que esta no es la única forma que conocemos hasta el momento para crear objetos, entonces podríamos tener 3 formas distintas en cuenta :

- Objetos literales : { }
- Operador new : new Object( )
- Método create : Object.create( )

MÉTODO CREATE

ECMAScript 5 define un método, Object.create (), que crea un nuevo objeto, utilizando su primer argumento como el prototipo de ese objeto. Object.create () también toma una opción
segundo argumento que describe las propiedades del nuevo objeto.
Object.create () es una función estática, no un método invocado en objetos individuales. A Úselo, simplemente pase el objeto prototipo deseado:

```
var o1 = Object.create({x:1, y:2});
```

Puede pasar null para crear un nuevo objeto que no tenga un prototipo, pero si lo hace esto, el objeto recién creado no heredará nada, ni siquiera los métodos básicos como toString () (lo que significa que tampoco funcionará con el operador +):

```
var o2 = Object.create (null);
```

Si desea crear un objeto vacío ordinario (como el objeto devuelto por {} o nuevo Object ()), pase Object.prototype:

```
var o3 = Object.create (Object.prototype);
// o3 es como {} o new Object ().
```

La capacidad de crear un nuevo objeto con un prototipo arbitrario (dicho de otra manera: el la capacidad de crear un "heredero" para cualquier objeto) es poderosa, y podemos simularla en ECMAScript 3 con una función como la del siguiente ejemplo:

```
// heredar() devuelve un objeto recién creado que hereda propiedades del
// objeto prototipo p. Utiliza la función Object.create () de ECMAScript
5 si
// está definida, y de lo contrario vuelve a una técnica anterior.
función heredar (p) {
  if (p == null) lanza TypeError (); // p debe ser un objeto no nulo
  if (Object.create) // Si Object.create () está definida ...
  return Object.create (p); // entonces úsala.
  var t = typeof p; // De lo contrario, haga un poco más de comprobación
de tipo
  if (t! == "object" && t! == "function") throw TypeError ();
  función f () {}; // Define una función de constructor ficticio.
  f.prototype = p; // Establezca su propiedad de prototipo en p.
  devolver nuevo f (); // Usa f () para crear un "heredero" de p.
```

```
}
```

Los objetos de JavaScript tienen un conjunto de "propiedades propias" y también heredan un conjunto de propiedades de su objeto prototipo. Para entender esto, debemos considerar el acceso a la propiedad en más detalle.

ACCESO A PROPIEDADES

Supongamos que consulta la propiedad **x** en el objeto **o**. Si **o** no tiene una propiedad propia con ese nombre, el objeto prototipo de **o** es consultado por la propiedad x. Si el prototipo objeto no tiene una propiedad propia con ese nombre, pero tiene un prototipo en sí mismo, la consulta se realiza en el prototipo del prototipo. Esto continúa hasta que la propiedad **x** es encontrado o hasta que se busque un objeto con un prototipo nulo. Como puedes ver, el prototipo atributo de un objeto crea una cadena o lista vinculada de la cual las propiedades son heredadas.

```
var o = {} // o hereda métodos de objeto de Object.prototype
o.x = 1; // y tiene una propiedad propia x.
var p = heredar (o); // p hereda propiedades de o y Object.prototype
p.y = 2; // y tiene una propiedad propia y.
var q = heredar (p); // q hereda propiedades de p, o, y Object.prototype
q.z = 3; // y tiene una propiedad propia z.
var s = q.toString (); // toString se hereda de Object.prototype
q.x + q.y // => 3: x y y se heredan de o y p
```

Ahora suponga que asigna a la propiedad **x** del objeto **o**. Si **o** ya tiene una propia (no heredada) propiedad llamada **x**, entonces la asignación simplemente cambia el valor de esta propiedad existente. De lo contrario, la asignación crea una nueva propiedad llamada **x** en el objeto **o**. Si **o** anteriormente heredó la propiedad **x**, esa propiedad heredada ahora está oculta por la propiedad propia recién creada con el mismo nombre.

La asignación de propiedades examina la cadena de prototipos para determinar si la asignación está permitida. Si **o** hereda una propiedad de solo lectura llamada **x**, por ejemplo, entonces la la asignación no está permitida. Si la asignación está permitida, sin embargo, siempre crea o establece una propiedad en el original objeto y nunca modifica la cadena del prototipo. El hecho de que la herencia ocurre en el momento de consultar propiedades pero no al configurarlas es una característica clave de JavaScript porque nos permite anular selectivamente las propiedades heredadas:

```
var unitcircle = {r: 1}; // Un objeto para heredar de
var c = inherit (unitcircle); // c hereda la propiedad r
c.x = 1; c.y = 1; // c define dos propiedades propias
c.r = 2; // c anula su propiedad heredada
```

```
unidadcircle.r; // => 1: el objeto prototipo no se ve afectado
```

Hay una excepción a la regla de que una asignación de propiedad falla o crea o establece una propiedad en el objeto original. Si **o** hereda la propiedad **x,** y esa propiedad es una propiedad de acceso con un método setter, luego se llama ese método setter en lugar de crear una nueva propiedad **x** en **o**. Tenga en cuenta, sin embargo, que el método setter es invocado en el objeto **o**, no en el objeto prototipo que define la propiedad, por lo que si el método setter define cualquier propiedad, lo hará en **o**, y volverá a dejar la cadena de prototipos sin modificar.

No es un error consultar una propiedad que no existe. Si la propiedad **x** no se encuentra como una propiedad propia o una propiedad heredada de **o**, la expresión de acceso a la propiedad **o.x** evalúa a **undefined**.

Sin embargo, es un error intentar consultar una propiedad de un objeto que no existe. Los valores nulos e indefinidos no tienen propiedades, y es un error consultar propiedades de estos valores:

```
// Crea una excepción TypeError. indefinido no tiene una propiedad de longitud
var libro = { }
var len = libro.subtitulo.length;
```

A menos que esté seguro de que tanto el libro como libro.subtitulo son (o se comportan como) objetos, no debe escribir la expresión libros.subtitulo.length, ya que podría generar una excepción. Aquí hay dos maneras de protegerse contra este tipo de excepción:

```
// Una técnica detallada y explícita
var len = undefined;
if (libro) {
 if (libro.subtitulo) len = libro.subtitulo.length;
}
// Una alternativa concisa e idiomática para obtener la longitud de los subtítulos o indefinido
var len = libro && libro.subtitulo && libro.subtitulo.length;
```

Intentar establecer una propiedad en null o undefined también provoca un TypeError, por supuesto. Los intentos de establecer propiedades en otros valores tampoco siempre tienen éxito: algunas propiedades son de solo lectura y no se pueden configurar, y algunos objetos no permiten la adición de nuevas propiedades. Curiosamente, sin embargo, estos intentos fallidos de establecer propiedades generalmente fallan silenciosamente:

```
// Las propiedades prototipo de los constructores incorporados son de solo lectura.
Object.prototype = 0;
// La asignación falla silenciosamente; Object.prototype sin cambios
```

Esta peculiaridad histórica de JavaScript se rectifica en el modo estricto de ECMAScript 5. En modo estricto, cualquier intento fallido de establecer una propiedad arroja una excepción TypeError.

Las reglas que especifican cuándo una asignación de propiedad tiene éxito y cuándo falla son intuitivas pero difícil de expresar concisamente. Un intento de establecer una propiedad p de un objeto **o** falla en estas circunstancias:

- **o** tiene una propiedad propia **p** que es de solo lectura: no es posible establecer propiedades de solo lectura.
- **o** tiene una propiedad heredada **p** que es de solo lectura: no es posible ocultar un elemento heredado de solo lectura con una propiedad propia del mismo nombre.
- **o** no tiene una propiedad propia **p**; **o** no hereda una propiedad **p** con un método setter y el atributo extensible de **o** es falso.

DEFINICIÓN DE PROPIEDADES

Hemos dicho que una propiedad de objeto es un nombre, un valor y un conjunto de atributos. En ECMAScript 5 el valor puede ser reemplazado por uno o dos métodos, conocidos como *getter* y *setter*. Las propiedades definidas por *getters* y *setters* a veces se conocen como propiedades accesor para distinguirlos de las propiedades de datos que tienen un valor simple.

Cuando un programa consulta el valor de una propiedad de accesor, JavaScript invoca el getter (sin pasar argumentos). El valor de retorno de este método se convierte en el valor de la expresión de acceso a la propiedad. Cuando un programa establece el valor de una propiedad de acceso, JavaScript invoca el método setter, pasando el valor del lado derecho de la asignación. Este método es responsable de "establecer", en cierto sentido, el valor de la propiedad. El valor de retorno del método setter se ignora.

Las propiedades de acceso no tienen un atributo escribible como propiedades de datos. Si una propiedad tiene ambos métodos, getter y setter, es una propiedad de lectura / escritura. Si solo tiene un getter, es una propiedad de solo lectura. Y si solo tiene un método setter, es solo de escritura (algo que no es posible con las propiedades de datos) e intentar leerlo

siempre evaluar a undefined.

La forma más fácil de definir las propiedades de acceso es con una extensión al objeto literal:

```
var o = {
// Una propiedad ordinaria
data_prop: value,
// una propiedad accessor definida como pares de
funciones
get accessor_prop() { /* function body here */ },
set accessor_prop(value) { /* function body here */ }
};
```

Las propiedades de acceso se definen como una o dos funciones cuyo nombre es el mismo que el nombre de propiedad, y con la palabra clave *function* reemplazada por *get* y / o *set*. Tenga en cuenta que no se usa dos puntos para separar el nombre de la propiedad de las funciones que tienen acceso a esa propiedad, pero que aún se necesita una coma después del cuerpo de la función para separar el método del siguiente método o propiedad de datos.

Como ejemplo, considere el siguiente objeto que representa un punto cartesiano 2D. Tiene propiedades de datos ordinarios para representar las coordenadas X e Y del punto, y tiene propiedades de acceso para el equivalente coordenadas polares del punto:

```
var p = {
 // x e y son propiedades de datos de lectura y escritura
regulares.
 x: 1.0,
 y: 1.0,
 // r es una propiedad de acceso de lectura-escritura con getter y
setter.
 // No olvides poner una coma después de los métodos de acceso.
 get r () {return Math.sqrt (this.x * this.x + this.y * this.y);
},
 set r (newvalue) {
 var oldvalue = Math.sqrt (this.x * this.x + this.y * this.y);
 var ratio = newvalue / oldvalue;
 this.x * = ratio;
 this.y * = ratio;
 },
 // theta es una propiedad de acceso de solo lectura con getter
```

```
solamente.
 get theta () {return Math.atan2 (this.y, this.x); }
};
```

Tenga en cuenta el uso de la palabra clave **this** en getters y setter arriba. JavaScript invoca estas funciones como métodos del objeto sobre el que están definidas, lo que significa que dentro del cuerpo de la función esto se refiere al objeto puntual. Entonces el método getter para la propiedad r puede hacer referencia a las propiedades **x** e **y** como **this.x** y **this.y**.

Las propiedades de acceso se heredan, al igual que las propiedades de datos, por lo que puede usar el objeto **p** definido anteriormente como un prototipo para otros puntos. Puedes darles a los nuevos objetos sus propias propiedades **x** e **y**, y heredarán las propiedades **r** y **theta**:

```
var q = heredar (p);
// Crea un nuevo objeto que hereda getters y setters
q.x = 0, q.y = 0; // Crea las propias propiedades de datos de q
console.log (q.r); // Y usa las propiedades de acceso heredadas
console.log (q.theta);
```

ATRIBUTOS DE PROPIEDADES

Además de un nombre y un valor, las propiedades tienen atributos que especifican si puede escribirse, enumerarse y configurarse. En ECMAScript 3, no hay forma de establecer estos atributos: todas las propiedades creadas por los programas de ECMAScript 3 son editables, enumerables, y configurable, y no hay forma de cambiar esto.

Para los fines de esta sección, vamos a considerar los métodos getter y setter de una propiedad de acceso como atributos de propiedad. Siguiendo esta lógica, incluso diremos que el valor de una propiedad de datos también es un atributo. Por lo tanto, podemos decir que una propiedad tiene un nombre y cuatro atributos. Los cuatro atributos de una propiedad de datos son **value**, **writable**, **enumerable** y **configurable**. Las propiedades de acceso no tienen un atributo de valor o una atributo de escritura: su capacidad de escritura está determinada por la presencia o ausencia de un setter. Entonces, los cuatro atributos de una propiedad de acceso son **get**, **set**, **enumerable** y **configurable**.

Las propiedades **writable**, **enumerable** y **configurable** son valores booleanos, y **get** y **set** son valores funciones, por supuesto.

Para obtener el descriptor de propiedad para una propiedad con nombre de un objeto especificado, llame Object.getOwnPropertyDescriptor ():

```
// Devuelve {valor: 1, escribible: verdadero, enumerable:
verdadero, configurable: verdadero}
Object.getOwnPropertyDescriptor ({x: 1}, "x");
// Devuelve indefinido para propiedades heredadas y propiedades
que no existen.
Object.getOwnPropertyDescriptor ({}, "x"); // undefined, no tal
prop
Object.getOwnPropertyDescriptor ({}, "toString"); // indefinido,
heredado
```

Ya hemos visto el método Object.create () y prendimos allí que el el primer argumento para ese método es el objeto prototipo para el objeto recién creado. Este método también acepta un segundo argumento opcional, que es el mismo que el segundo argumento a Object.defineProperties (). Si pasa un conjunto de descripciones de propiedad a Object.create (), luego se usan para agregar propiedades al objeto recién creado.

Object.defineProperty () y Object.defineProperties () arrojan TypeError si el intento para crear o modificar una propiedad no está permitido. Esto sucede si intentas agrega una nueva propiedad a un objeto no extensible. El atributo writable rige los intentos de cambiar el atributo de valor y el atributo configurable rige los intentos de cambiar los otros atributos (y también especifica si un propiedad puede ser eliminada). Sin embargo, las reglas no son completamente sencillas. Es posible cambiar el valor de una propiedad no escribible si esa propiedad es configurable, por ejemplo. Además, es posible cambiar una propiedad de escribible a no escribible incluso si esa propiedad no es configurable. Aquí están las reglas completas. Llamadas a Object.defineProperty () o Object.defineProperties () que intentan lanzar TypeError:

- Si un objeto no es extensible, puede editar sus propias propiedades existentes, pero no puede agregar nuevas propiedades
- Si una propiedad no es configurable, no puede cambiar sus atributos configurable o enumerable.
- Si una propiedad de acceso no es configurable, no puede cambiar su getter o setter y no puede cambiarlo a una propiedad de datos.
- Si una propiedad de datos no es configurable, no puede cambiarla a una propiedad de acceso.
- Si una propiedad de datos no es configurable, no puede cambiar su atributo de escritura de falso a verdadero, pero puede cambiarlo de verdadero a falso.
- Si una propiedad de datos no es configurable y no se puede escribir, no puede cambiar su valor.

Puede cambiar el valor de una propiedad que es configurable pero no se puede escribir, sin embargo , porque eso sería lo mismo que hacer que se pueda escribir, luego cambiar el valor, luego convirtiéndolo de nuevo a no escribible.

Para entender la siguiente forma de creación de nuevos objetos bajo el operador new vamos a tener que tener además un conocimiento adicional sobre las funciones, ya que estas representan el núcleo del siguiente tema.

FUNCIONES

Además de muchas de las cosas que podríamos mencionar sobre funciones, nos vamos a focalizar en los siguientes aspectos para tener un mejor entendimiento de las mismas :

- Son funciones de primer orden
- Son funciones variadicas
- Tienen ámbito
- Tienen contexto

PRIMER ORDEN

Funciones de orden superior son funciones que pueden tomar otras funciones como argumentos o devolverlos como resultados. En cálculo , un ejemplo de una función de orden superior es el operador diferencial d / dx , que devuelve la derivada de una función f .

Las funciones de orden superior están estrechamente relacionadas con las funciones de primera clase en las cuales las funciones de orden superior y las funciones de primera clase pueden recibir como argumentos y resultados otras funciones. La distinción entre los dos es sutil: "de orden superior", describe un concepto matemático de funciones que operan sobre otras funciones, mientras que la "primera clase" es un término informático que describe las entidades del lenguaje de programación que no tienen ninguna restricción de su utilización (por lo tanto funciones de primera clase pueden aparecer en cualquier parte del programa que otras entidades de primer nivel como los números pueden, incluidos como argumentos a otras funciones y como sus valores de retorno) [3].

FUNCIONES VARIADICAS

En matemáticas y en programación de computadoras, una función variadica es una función de aridad indefinida, es decir, una que acepta un número variable de argumentos. El soporte para funciones variadic difiere ampliamente entre los lenguajes de programación [4] [5].

AMBITO

El ámbito de una función(scope) podría ser definido como el alcance que tiene la misma sobre variables dentro del programa en ejecución. El mismo en Javascript tiene un alcance

similar que en otros lenguajes de programación desde el punto de vista en que todas las funciones pueden tener acceso a variables globales y locales(pre inicializadas o declaradas dentro del cuerpo de la función) pero a esto le sumamos el concepto de **closure.**

CLOSURE

Al igual que la mayoría de los lenguajes de programación modernos, JavaScript usa el alcance léxico. Esto significa que las funciones se ejecutan usando el alcance variable que estaba en efecto cuando(ellas mismas, las funciones) eran definidas, no el ámbito de la variable que está en vigencia cuando se invocan. A fin de que implementar el alcance léxico, el estado interno de un objeto de función de JavaScript debe incluir no solo el código de la función sino también una referencia a la cadena de alcance actual. Esta combinación de un objeto de función y un alcance (un conjunto de enlaces de variables) en el que se resuelven las variables de la función se llama closure en la literatura de ciencias de la computación.

Técnicamente, todas las funciones de JavaScript son closures: son objetos y tienen un alcance cadena asociada a ellos. La mayoría de las funciones se invocan utilizando la misma cadena de alcance que estaba en efecto cuando se definió la función, y realmente no importa que haya un closure involucrado. Los closures se vuelven interesantes cuando se invocan bajo una cadena de alcance diferente a la que estaba en efecto cuando se definieron. Esta ocurre más comúnmente cuando se devuelve un objeto de función anidada de la función dentro de la cual fue definida. Hay una serie de potentes técnicas de programación que involucran este tipo de closures de funciones anidadas, y su uso se ha vuelto relativamente común en la programación de JavaScript. Los closures pueden parecer confusos cuando te encuentras por primera vez con ellos, pero es importante que los entiendas lo suficientemente bien como para usarlos cómodamente. Podríamos analizar el siguiente ejemplo :

```
var a = 1
function externa(x){
     var b = 10
     console.log(x,b,a)
}
```

Como podemos apreciar, la función externa puede hacer uso de la variable global a, la pre inicializada x y local b. Estamos acostumbrados a este comportamiento por lo cual no debería ser de sorpresa. Recordemos además que las funciones pueden contener otras funciones en su interior y hasta incluso retornarlas como su valor :

```
var a = 1
```

```
function externa(x){
    var b = 10
    console.log(x,b,a)
    return function interna(y){
        console.log(y)
    }
}
var foo = externa(20)
```

Ahora estamos devolviendo una función como valor de la ejecución de la función externa y lo estamos guardando en la variable foo. De esta forma , foo, pasa a valer una función, particularmente la función interna, con lo cual podemos ejecutar la variable foo.
Tengamos en cuenta que luego de que una función se ejecuta, todas las variables locales de la misma son destruidas por el Garbage Collector, el algoritmo que se encarga de “limpiar” nuestra memoria para no sobrecargar el programa. Es decir que a partir de la creación de la variable foo y su posterior asignación, la variable **b** y **x** de externa ya dejaron de existir. Si esto es cierto, entonces cómo puede ser posible que la función interna pueda adoptar el siguiente comportamiento :

```
var a = 1
function externa(x){
    var b = 10
    console.log(x,b,a)
    return function interna(y){
        console.log(y + x)
    }
}

var foo = externa(20)

console.log(foo)
//La ejecución de foo nos suma el valor original de x ya
que mantiene su valor gracias al closure generado
foo(15) //35
```

Es importante entonces entender el funcionamiento de cuando un closure es creado y hasta incluso cerrado. Consideremos el siguiente ejemplo :

```
// Esta función retorna una función que siempre retorna v
function constfunc(v) {
    return function() {
        return v;
    };
}
// Creamos un array de funciones constantes :
var funcs = [];
for(var i = 0; i < 10; i++)
    funcs[i] = constfunc(i);
// La función guardada en el elemento 5 del array retorna su valor
5.
funcs[5]() // => 5
```

Si no tuviéramos este conocimiento podríamos caer en el error común de optar por un código como el siguiente :

```
// Retorna un array de funciones que retornan un valor del 0 al 9
function constfuncs() {
    var funcs = [];
    for(var i = 0; i < 10; i++)
        funcs[i] = function() {
            return i;
        };
    return funcs;
}

var funcs = constfuncs();
funcs[5]() // Que retorna?
```

El código anterior, aunque demasiado similar al del ejemplo previo, crea 10 closuresy los almacena en una matriz. Los closures son todos definido dentro de la misma invocación de la función, para que compartan el acceso a la variable **i**. Cuando devuelve constfuncs (), el valor de la variable **i** es 10 y los 10 closures comparten este valor. Por lo tanto, todas las funciones en la matriz devuelta de funciones devuelven el mismo valor, que no es lo que queríamos en absoluto. Es importante recordar que el alcance cadena asociada con un closure es "en vivo". Las funciones anidadas no hacen copias privadas del alcance ni hacen copias estáticas de los enlaces de variable.

CONTEXTO

El contexto de una función es usualmente determinado por cómo la función se ejecuta y en líneas generales podríamos decir que es aquel objeto que al cual la función pertenece. El contexto de una función es referenciado por la palabra clave **this.**

JavaScript es un lenguaje de un solo hilo, lo que significa que solo se puede ejecutar una tarea a la vez. Cuando el intérprete de JavaScript ejecuta código inicialmente, primero entra en un contexto de ejecución global de manera predeterminada. Cada invocación de una función desde este punto dará como resultado la creación de un nuevo contexto de ejecución.

Aquí es donde a menudo se establece la confusión, el término "contexto de ejecución" se refiere en realidad a todos los efectos, refiriéndose más al alcance y no al contexto como se discutió anteriormente. Es una desafortunada convención de nombres, sin embargo, es la terminología definida por la especificación ECMAScript, por lo que estamos un poco atascados con ella.

Cada vez que se crea un nuevo contexto de ejecución, se anexa a la parte superior de la pila de ejecución. El navegador siempre ejecutará el contexto de ejecución actual que está encima de la pila de ejecución. Una vez completado, se eliminará de la parte superior de la pila y el control volverá al siguiente contexto de ejecución.

Un contexto de ejecución se puede dividir en una fase de creación y ejecución. En la fase de creación, el intérprete primero creará un objeto variable (también llamado objeto de activación) que se compone de todas las variables, declaraciones de funciones y argumentos definidos dentro del contexto de ejecución. A partir de ahí, la cadena del alcance se inicializa a continuación, y el valor de esto se determina al final. Luego, en la fase de ejecución, el código es interpretado y ejecutado.

Dado que el contexto de una función no es estático podríamos estudiar los siguientes ejemplos de ejecución donde el contexto varía :

```
//window.globalCtx = function(){}
 function globalCtx(){
     console.log(this)
}
//window.globalCtx()
globalCtx() //window{...}
```

En este primer ejemplo podemos observar cómo el contexto “natural” de una función global apunta al objeto global window, dado que dicha función “le pertenece”, ergo su contexto es él mismo.

```
var persona = {
     nombre : "Juan",
     saludo : function(){
          console.log(this)//{nombre:"Juan"...}
          var self = this
          setTimeout(function(){
              console.log(this)//window
              console.log(self)//{nombre:"Juan"...}
          },0)
     }
}

persona.saludo()
```

En el ejemplo anterior podemos observar como invocando al método saludo del objeto persona, su contexto nos muestra al mismo objeto persona, ya que el método le pertenece. 0(cero) segundos más tarde, intentamos realizar la misma operación, con la diferencia que ahora obtenemos otro resultado. Esto se debe a que el contexto de ejecución de la función setTimeout dentro de su callback es window, ya que ambas dos pertenecen a window. Para evitar este comportamiento podemos guardar nuestro contexto original en otra variable la cual podemos usar posteriormente.

APPLY vs CALL

Las funciones de JavaScript son objetos y, como todos los objetos de JavaScript, tienen métodos. Dos de estos métodos, call () y apply () invocan indirectamente la función. Ambos métodos le permite especificar explícitamente este valor para la invocación, lo que significa que puede invocar cualquier función como método de cualquier objeto, incluso si no es realmente un método de ese objeto Ambos métodos también le permiten especificar los argumentos para la invocación. El método call () usa su propia lista de argumentos como argumentos para la función y el el método apply () espera que una matriz de valores se use como argumentos.

```
function foo(a,b){
     console.log(this)
    console.log(a,b)
}
foo(1,2) //window{...} 1 2
foo.apply({contexto:"apply"},[10,20]) //
```

```
{contexto:"apply"} 10 20
foo.call({contexto:"call"},100,200) //{contexto:"call"}
100 200
```

OPERADOR NEW

El nuevo operador crea e inicializa un nuevo objeto. La nueva palabra clave debe ser seguida por una invocación de función. Una función utilizada de esta manera se llama constructor y sirve para inicializar un objeto recién creado. El núcleo de JavaScript incluye constructores incorporados para tipos nativos. Por ejemplo:

```
var o = new Object(); // Crea un objeto vacio. Mismo que {}.
var a = new Array(); // Crea un array vacio. Mismo que [].
var d = new Date(); // Crea un objeto con la fecha actual
var r = new RegExp("js"); // Crea un objeto de expresión regular
```

Usemos como ejemplo el siguiente constructor :

```
function Foo(){
}
```

Cuando se ejecuta la función asociada al operador new suceden las siguientes cosas:

1. Se crea un nuevo objeto que hereda de Foo.prototype.
2. La función del constructor Foo se llama con los argumentos especificados, y con esto enlazado al objeto recién creado. New Foo equivale a newFoo (), es decir, si no se especifica ninguna lista de argumentos, se llama a Foo sin argumentos.
3. El objeto devuelto por la función constructora se convierte en el resultado de la nueva expresión completa. Si la función constructora no devuelve explícitamente un objeto, en su lugar se usa el objeto creado en el paso 1. (Normalmente, los constructores no devuelven un valor, pero pueden optar por hacerlo si desean anular el proceso normal de creación de objetos).

De donde:

```
new Foo()
//Ejecutar new Foo() podría verse también de la siguiente manera :
var buffer = {}
Foo.call(buffer)
```

```
return buffer
//Ya que creamos un objeto nuevo, ejecutamos la función
asignándole el contexto con el nuevo objeto creado y terminamos
retornando dicho objeto.
```

Dado que el contexto de ejecución de la función Foo, o a fines prácticos cualquier función, va a ser un objeto nuevo y vacío, podemos usar a las funciones como creadoras de “clases”. Es por eso que les solemos decir a estas funciones : **funciones constructoras.**

FUNCIONES CONSTRUCTORAS

Un constructor es una función diseñada para la inicialización de objetos creados recientemente. Se invocan constructores utilizando la palabra clave new como se describe en anteriormente. Invocaciones de constructor utilizando new automáticamente crea el nuevo objeto, por lo que el constructor solo necesita inicializar el estado de ese nuevo objeto. La característica fundamental de las invocaciones de constructores es que la propiedad prototype del constructor se usa como el prototipo del nuevo objeto. Esto significa que todos los objetos creados con el mismo constructor heredan del mismo objeto y por lo tanto son miembros de la misma clase. Consideremos entonces el siguiente ejemplo :

```
function Rango(desde, hasta) {
    // Guardamos los puntos de comienzo y fin.
    // Son propiedades no heredadas, únicas para el objeto.
    this.desde = desde;
    this.hasta = hasta;
}

Rando.prototype = {
    // Retorna true si x se encuentra en el rango, falso de lo
contrario
    includes: function(x) {
        return this.from <= x && x <= this.to;
    }
 };

var r = new Range(1,3);
r.includes(2); //true
```

1. https://developer.mozilla.org/es/docs/Web/JavaScript/Herencia_y_la_cadena_de_protipos
2. https://developer.mozilla.org/es/docs/Web/JavaScript/Guide/Details_of_the_Object_Model
3. https://es.wikipedia.org/wiki/Programaci%C3%B3n_funcional
4. https://en.wikipedia.org/wiki/Variadic_function
5. https://jherax.wordpress.com/2014/07/06/javascript-funciones-variadicas/