DigitalHouse >
Coding School

INTRODUCCIÓN AL ASINCRONISMO

“

En programación el **asincronismo** consiste en poder iniciar una **acción** sin depender de la **finalización** de acciones anteriores.

De esta manera nuestro programa puede seguir avanzando sin esperar que cada acción termine.

1.

LA PILA DE TAREAS

CÓMO FUNCIONA JAVASCRIPT

Antes de poder hablar de asincronismo, necesitamos entender un poco más cómo funciona Javascript por dentro.

Lo primero que necesitamos saber, es que por sí solo **Javascript** es un lenguaje que tiene **un solo hilo de ejecución**.

En términos simples eso quiere decir que **sólo puede hacer una cosa a la vez.**

Para determinar el orden en que se ejecutan las tareas, Javascript cuenta con algo que se conoce como la **Pila de Tareas**.

LA PILA DE TAREAS

Podemos imaginar la pila de tareas como un tubo de pelotas de tenis, donde cada pelota es una tarea que le encargamos a Javascript.

Al igual que un el tubo de pelotas de tenis, en la **pila de tareas** sólo podemos agregar tareas por encima de las que ya están. Y si queremos sacar una tarea, siempre va a ser la que esté más arriba.

Esta modalidad se la conoce como LIFO (Last In First Out).

LA PILA DE TAREAS

Si juntamos todo, tenemos a **Javascript** que lee nuestro código y interpreta las tareas.

Luego tenemos la **pila de tareas** que define el orden en el que se ejecutan nuestras tareas.

Por último sabemos que sólo se pueden ejecutar **una cosa a la vez**.

Veamos entonces cómo funciona esto con código real…


```
function saludar() {

      return '¡Hola!';

}

console.log(saludar());
```

Arranca nuestro programa, Javascript engloba nuestro código en una función llamada **main()**.

CÓDIGO

main()

PILA DE TAREAS

CONSOLA


```
function saludar() {

      return '¡Hola!';

}

console.log(saludar());
```

Declaración de una función, no hay tareas por acá.

CÓDIGO

main()

PILA DE TAREAS

CONSOLA


```
function saludar() {

      return '¡Hola!';

}

console.log(saludar());
```

Invocación de console.log(), es nuestra primera tarea.

Se agrega a la **pila**.

CÓDIGO

console.log(...)

main()

PILA DE TAREAS

CONSOLA

JS

```
function saludar() {

      return '¡Hola!';

}

console.log(saludar());
```

Invocación de saludar(), es una subtarea dentro de console.log().

Se agrega a la **pila**.

CÓDIGO

saludar()

console.log(...)

main()

PILA DE TAREAS

CONSOLA

JS

```
function saludar() {

      return '¡Hola!';

}

console.log(saludar());
```

Se ejecuta saludar(), que le retorna un valor a console.log().

Se va de la pila.

CÓDIGO

console.log(...)

main()

PILA DE TAREAS

CONSOLA


```
function saludar() {

      return '¡Hola!';

}

console.log(saludar());
```

Se ejecuta console.log(), que ahora tiene lo que retornó saludar().

El resultado se muestra por consola.

El **console.log()** se va de la pila.

CÓDIGO

main()

PILA DE TAREAS

```
'¡Hola!'
```

CONSOLA


```
function saludar() {

      return '¡Hola!';

}

console.log(saludar());
```

No hay más tareas, nuestro programa finalizó su ejecución.

La función **main()** se va de la pila.

CÓDIGO

PILA DE TAREAS

```
'¡Hola!'
```

CONSOLA

2.

EL NAVEGADOR Y LAS APIS WEB

CÓMO FUNCIONA JAVASCRIPT (Parte 2)

Si dijimos que Javascript por sí solo no puede hacer más de una cosa a la vez, entonces necesitamos de alguien más que se pueda encargar del resto de las tareas.

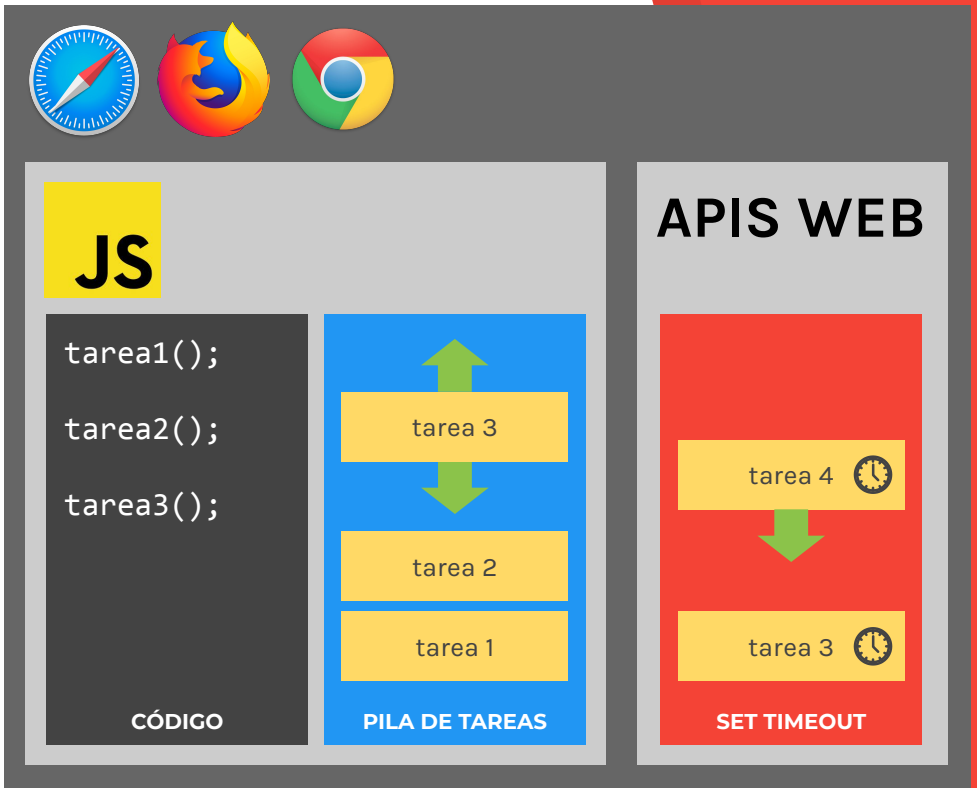
Ese alguien más en este caso es el **navegador**, que le agrega funcionalidad a Javascript a través de lo que se conocen como **APIs Web**.

EL NAVEGADOR Y LAS APIS WEB

Si la **pila de tareas** se encarga de ordenar las tareas y ejecutar una después de la otra.

Las **APIs WEB** se encargarán de todos aquellos **llamados** que no se puedan resolver en el momento, osea los **asíncronos**.

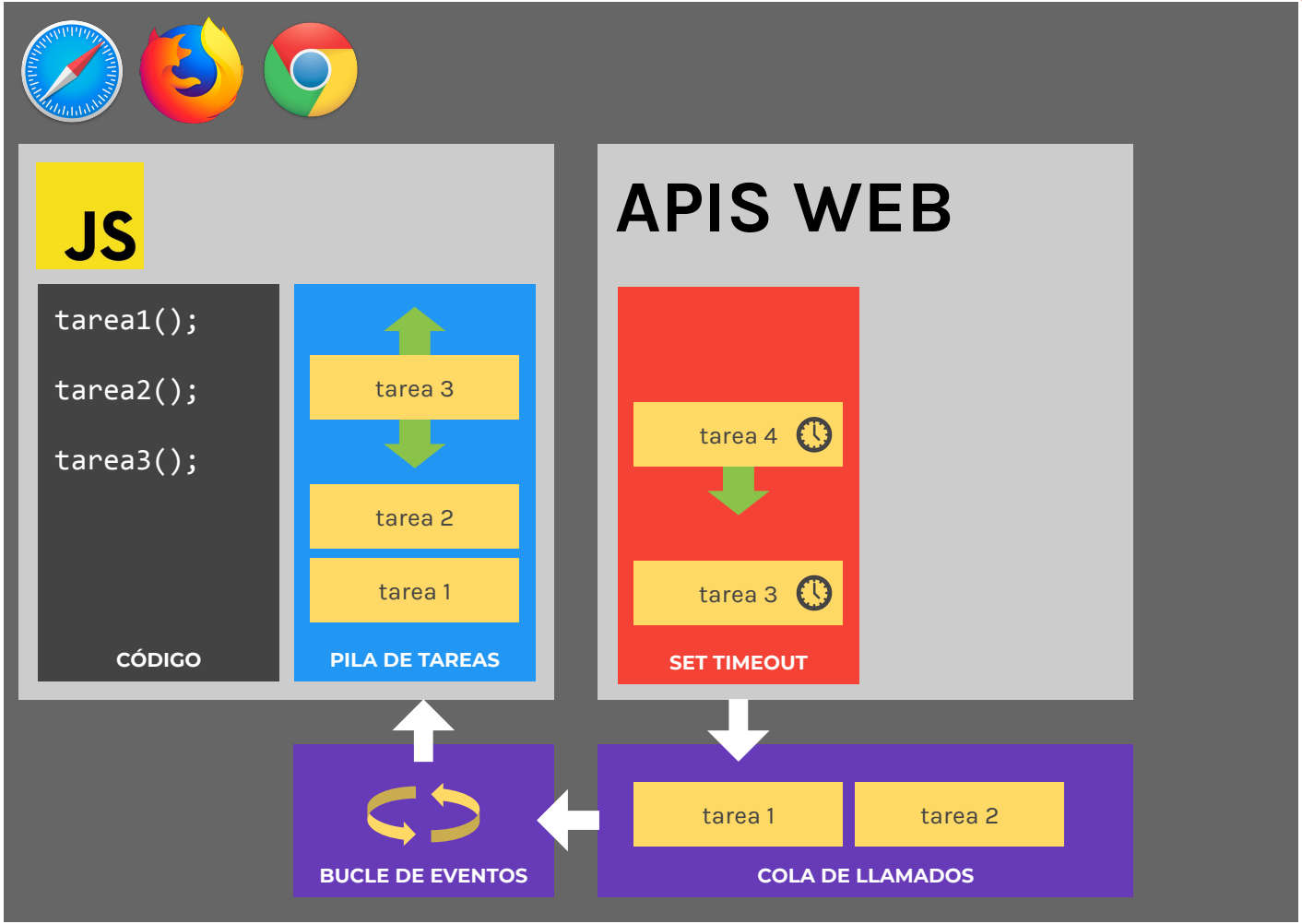
1. **Primero** se resolverá todo el **código sincrónico**.
2. **Luego** se resolverán los llamados a las **APIs WEB**



CÓMO FUNCIONA JAVASCRIPT

Código sincrónico primero

Código asincrónico después


JS
tarea1();
tarea2();
tarea3();
CÓDIGO
tarea 3
tarea 2
tarea 1
PILA DE TAREAS
APIS WEB
tarea 4
tarea 3
SET TIMEOUT
BUCLE DE EVENTOS
tarea 1
tarea 2
COLA DE LLAMADOS

CÓMO FUNCIONA JAVASCRIPT

Adicionalmente a la Pila de tareas, Javascript tiene algo que llamamos la **Cola de tareas**, que organiza los **llamados** a funciones por el **orden de llegada** y nunca pondrá una función por encima de otra.

Veamos un ejemplo de cómo funciona, usando una función nativa de JS: **setTimeout**.

setTimeout()

Recibe dos parámetros: el primero, un **callback** *(función)* y el segundo, un **número**, que representa la cantidad de milisegundos que deberán pasar antes de ejecutar el callback.

{}

```
setTimeout(function(){
    console.log('¡Ya pasó 1 segundo!');
}, 1000);
```

{ código }

```
function saludar() {
     return 'Hola!';
}
function despedirme() {
     return 'Ya pasaron 2 segundos, ¡Chau!';
}
setTimeout(function(){
     console.log(despedirme());
}, 2000);
console.log(saludar());
```

{ código }

```
function saludar() {
     return 'Hola!';
}
function despedirme() {
     return 'Ya pasaron 2 segundos, ¡Chau!';
}
setTimeout(function(){
     console.log(despedirme());
}, 2000);
console.log(saludar());
```

Lo primero en cargarse en la pila de tareas es el llamado al **console.log.**

EJEMPLO GRÁFICO

{ código }

```
function saludar() {
     return 'Hola!';
}
function despedirme() {
     return 'Ya pasaron 2 segundos, ¡Chau!';
}
setTimeout(function(){
     console.log(despedirme());
}, 2000);
console.log(saludar());
```

Dentro del console.log hay un **llamado** a la función **despedirme**, pero al estar dentro del setTimeout, deben pasar dos segundos antes de ejecutarse.

Por lo tanto, en lugar de ocupar un lugar en la Pila de tareas, la función **despedirme** irá a parar a la **Cola de tareas.**

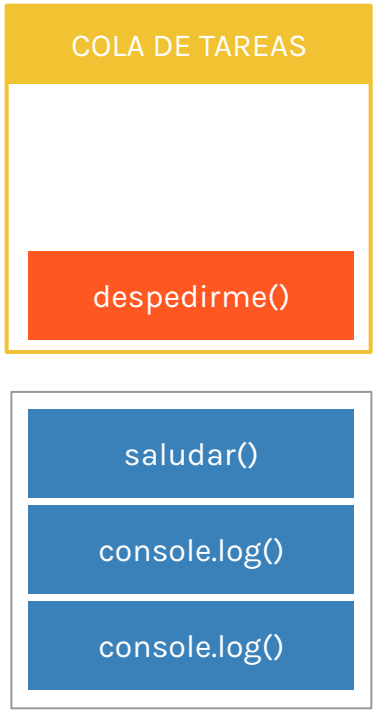
EJEMPLO GRÁFICO

{ código }

```
function saludar() {
     return 'Hola!';
}
function despedirme() {
     return 'Ya pasaron 2 segundos, ¡Chau!';
}
setTimeout(function(){
     console.log(despedirme());
}, 2000);
console.log(saludar());
```

La lectura del script continúa y lo próximo en ir a la **Pila de tareas** es el llamado al console.log seguido de la función saludar().

EJEMPLO GRÁFICO



{ código }

```
function saludar() {
    return 'Hola!';
}
function despedirme() {
    return 'Ya pasaron 2 segundos, ¡Chau!';
}
setTimeout(function(){
    console.log(despedirme());
}, 2000);
console.log(saludar());
```

Se **ejecuta** la función saludar, por lo tanto se va de la pila de tareas.

EJEMPLO GRÁFICO

{ código }

```
function saludar() {
     return 'Hola!';
}
function despedirme() {
     return 'Ya pasaron 2 segundos, ¡Chau!';
}
setTimeout(function(){
     console.log(despedirme());
}, 2000);
console.log(saludar());
```

Se ejecuta el console.log, que en esta instancia ya tiene el resultado que retornó la función saludar (*hola*), y se va de la pila de tareas.

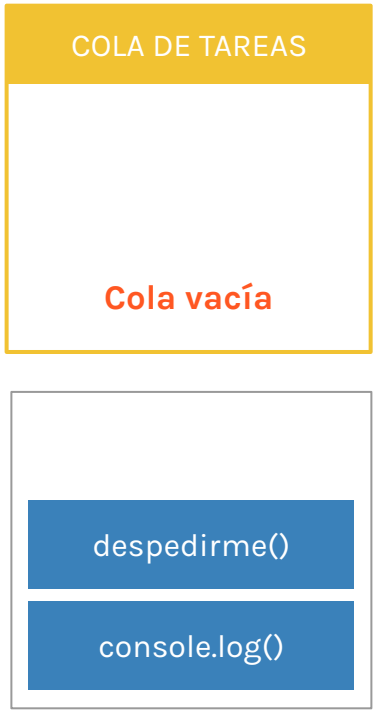
EJEMPLO GRÁFICO

{ código }

```
function saludar() {
     return 'Hola!';
}
function despedirme() {
     return 'Ya pasaron 2 segundos, ¡Chau!';
}
setTimeout(function(){
     console.log(despedirme());
}, 2000);
console.log(saludar());
```

Pasados dos segundos, se hace el llamado a la función **despedirme**. Por lo tanto, deja de estar en la Cola de tareas para pasar a la **Pila de tareas**.

EJEMPLO GRÁFICO

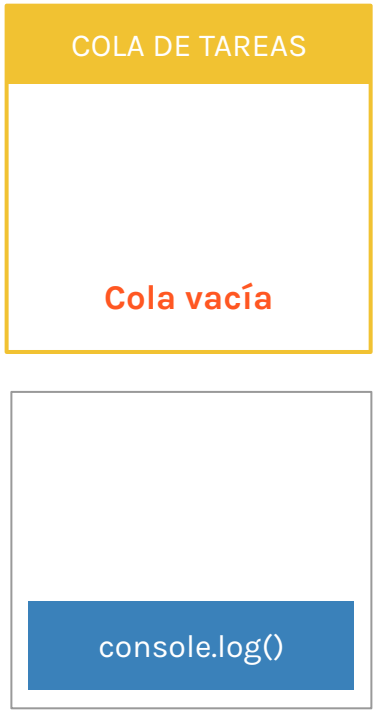


{ código }

```
function saludar() {
    return 'Hola!';
}
function despedirme() {
    return 'Ya pasaron 2 segundos, ¡Chau!';
}
setTimeout(function(){
    console.log(despedirme());
}, 2000);
console.log(saludar());
```

Se **ejecuta** la función **despedirme**, por lo tanto se va de la pila de tareas.

EJEMPLO GRÁFICO

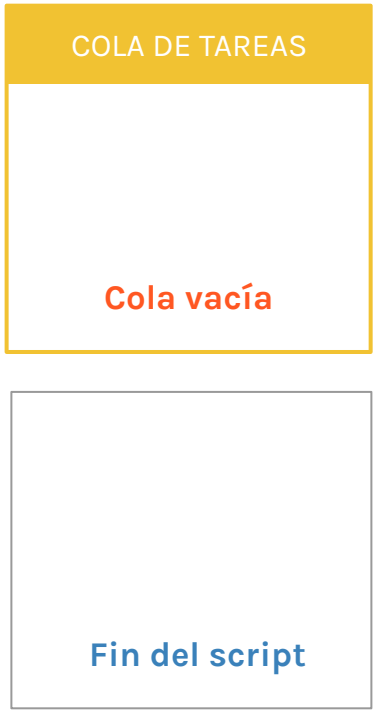


{ código }

```
function saludar() {
     return 'Hola!';
}
function despedirme() {
     return 'Ya pasaron 2 segundos, ¡Chau!';
}
setTimeout(function(){
     console.log(despedirme());
}, 2000);
console.log(saludar());
```

Se ejecuta el console.log, que en esta instancia ya tiene el string que retornó la función despedirme, y se va de la pila de tareas.

EJEMPLO GRÁFICO



Pero...¿cómo es esto posible si JavaScript es un lenguaje **single threaded**?

Para saber cómo lo hace, es necesario hablar acerca del **Event Loop.**

EVENT LOOP

Se encarga de monitorear qué funciones son ejecutadas y en qué momento. El mismo tiene la capacidad de saber cuándo nuestra **Pila de tareas** está **vacía**, y es recién en ese momento que comenzará a ejecutar las funciones que estén presentes en la **Cola de tareas**.

Es así como javascript genera el asincronismo, que es la posibilidad que presenta el lenguaje de realizar **múltiples tareas** en un **mismo** espacio de **tiempo**.