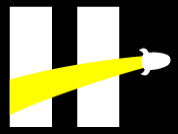
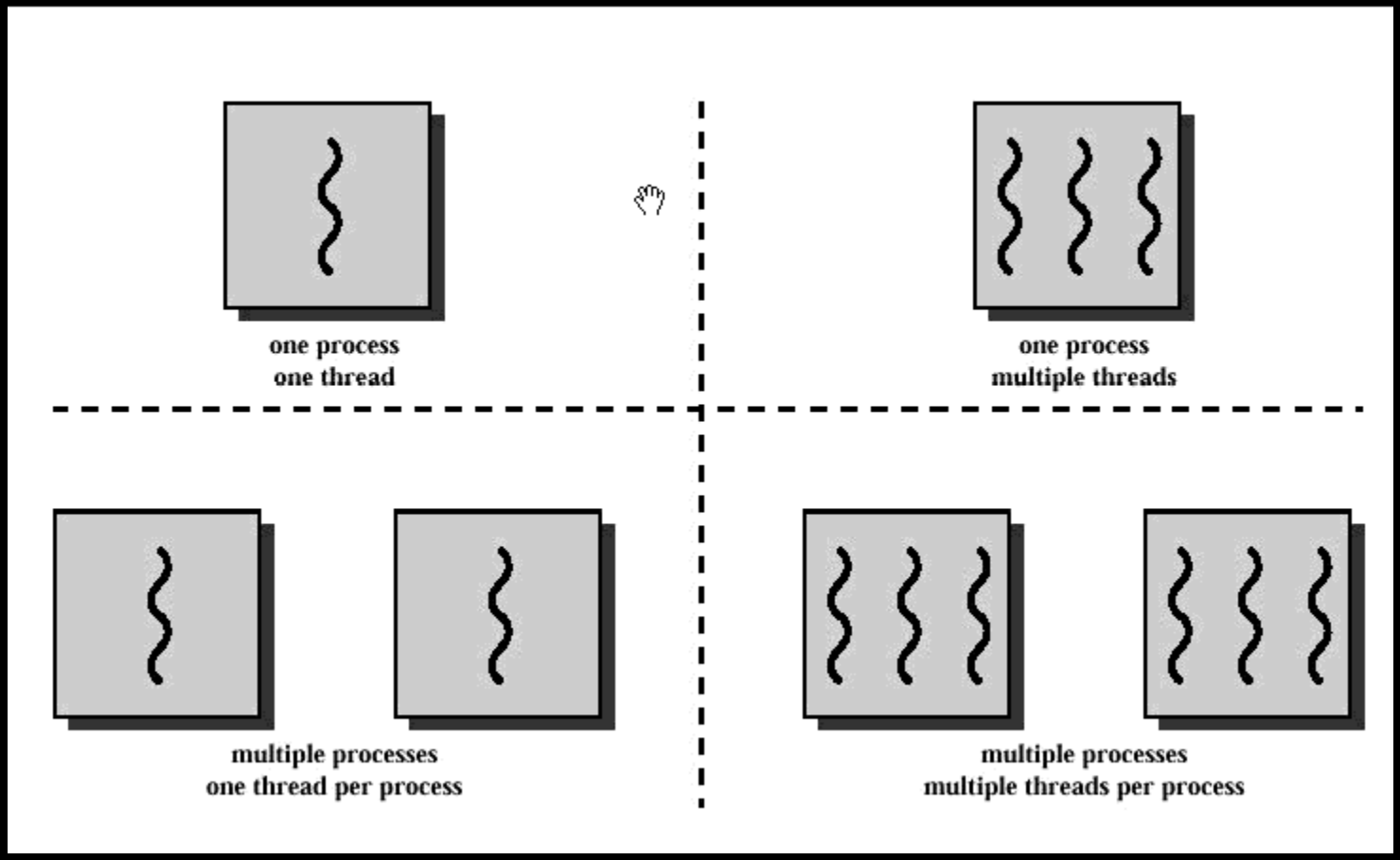
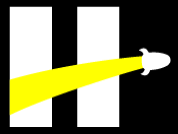

HENRY



Parte I:
Entiendiendo JS



Single Threaded y Sincrónico

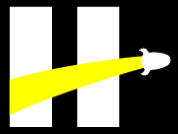


Syntax Parser

Lexical Enviroment

```
1 function hola(){
2         var foo = 'Hola!';
3 }
4
5 var bar = 'Chao';
```

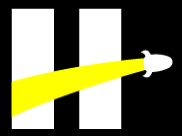
Por ejemplo, para el interprete las dos declaraciones de variable del arriba tendrán significados muy distintos. Si bien la operación es igual en los dos (asignación) al estar en lugares distintos (una dentro de una función y la otra no) el interprete las parseará de forma distinta



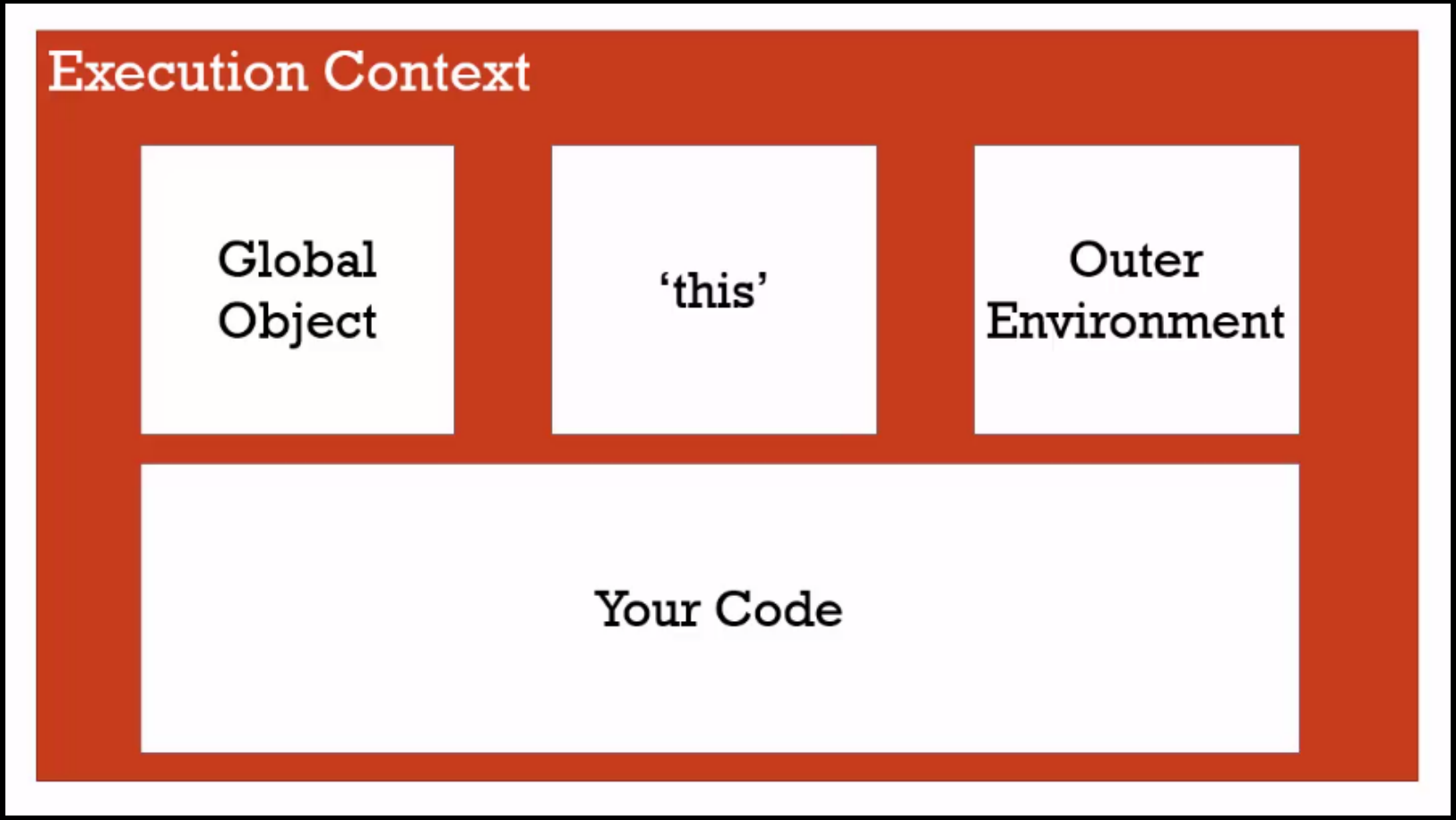
Execution Context

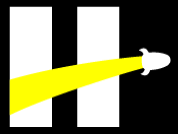
```
// global context

var sayHello = 'Hello';

function person() {          // execution context

    var first = 'David',
        last  = 'Shariff';

    function firstName() {   // execution context
        return first;
    }

    function lastName() {    // execution context
        return last;
    }

    alert(sayHello + firstName() + ' ' + lastName());

}
```

El contexto de ejecución contiene información sobre qué código se está ejecutando en cada momento. Además de mantener el código que tiene que ejecutar, también mantiene más información sobre de donde se invocó ese código, en qué lexical enviroment está, etc...



Execution Context

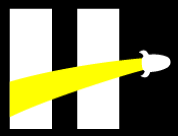




Hoisting

```
1 bar();
2 console.log(foo);
3
4 var foo = 'Hola, me declaro';
5 function bar() {
6         console.log('Soy una función');
7 }
```

El hosting es el primer ejemplo de las *cosas extras* que hace el interprete sin que nosotros se lo pidamos. Si no las conocemos, nos puede pasar que veamos comportamientos extraños y no sepamos de donde vienen (como que podamos usar funciones que no hemos declarado antes de invocarlas!!)



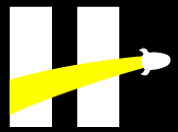
Execution Stack

```
1  function b() {
2    console.log('B!')
3  };
4
5  function a() {
6    // invoca a la función b
7    b();
8  }
9
10 //invocamos a
11 a();
```

b()
Execution Context
(create and execute)

a()
Execution Context
(create and execute)

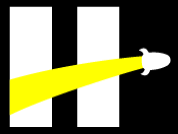
Global Execution Context
(created and code is executed)



Scope

```
1  var global = 'Hola!';
2
3  function a() {
4          // como no hay una variable llamada global en este contexto,
5          // busca en el outer que es el global
6          console.log(global);
7          global = 'Hello!'; // cambia la variable del contexto global
8  }
9
10 function b(){
11         // declaramos una variable global en nuestro contexto
12         // esta es independiente
13         var global = 'Chao';
14         console.log(global);
15 }
16
17 a(); // 'Hola!'
18 b(); // 'Chao'
19 console.log(global); // 'Hello'
```

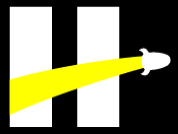
Para esto vamos a introducir el término scope, este es el set de variable, objeto y funciones al que tenemos acceso en determinado contexto.



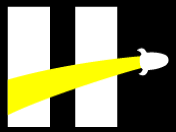
Scope

```
var global = 'Hola!';

function b(){
        var global = 'Chao';
        console.log(global); // Chao
        function a() {
                // como no hay una variable llamada global en este contexto,
                // busca en el outer que es scope de b;
                console.log(global); //Chao
                global = 'Hello!'; // cambia la variable del contexto de b()
        }
        a();
}

//a(); Ya no puedo llamar a a desde el scope global, acá no existe.
b();
console.log(global); // 'Hola!'
```

Cada contexto maneja sus propias variables,
y son independientes de los demás



Tipos de Datos



Static vs Dynamic Typing

Java

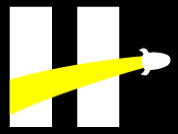
Static typing:

```
String name;        Variables have types
name = "John";      Values have types
name = 34;          Variables cannot change type
```

JavaScript

Dynamic typing:

```
var name;           Variables have no types
name = "John";      Values have types
name = 34;          Variables change type dynamically
```

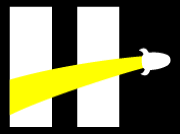


Operadores

```
1 var a = 2 + 3; // 5
2
3 function suma(a,b){
4         return a + b;
5         // usamos el mismo operador como ejemplo
6         // Si no deberiamos hacer sumas binarias!
7 }
8 var a = suma(2,3) // 5
```

Infix Expression	Prefix Expression	Postfix Expression
A + B	+ A B	A B +
A + B * C	+ A * B C	A B C * +

Un operador no es otra cosa que una función

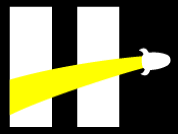


Precedencia de Operadores y Asociatividad

La *precedencia de operadores* es básicamente el orden en que se van a llamar las funciones de los operadores.

La *Asociatividad de operadores* es el orden en el que se ejecutan los operadores cuando tienen la misma precedencia, es decir, de izquierda a derecha o de derecha a izquierda.

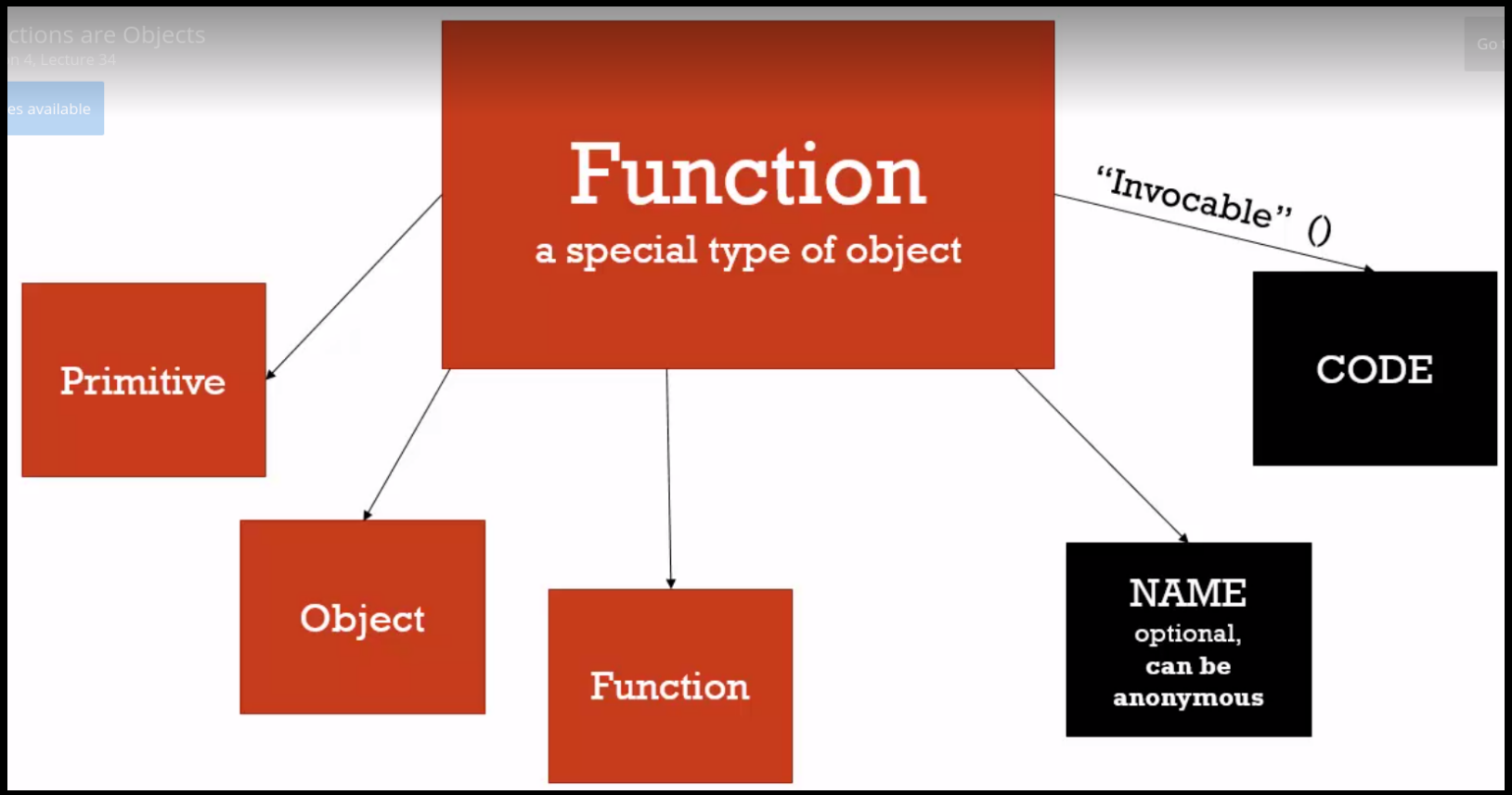
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precedence#Table



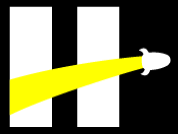
Coerción de Datos

```

Number('3') // devuelve el número 3. Obvio!
Number(false) // devuelve el número 0. mini Obvio.
Number(true)  // devuelve el número 1. menos mini Obvio.
Number(undefined) // devuelve `NaN`. No era obvio, pero tiene sentido.
Number(null) // devuelve el número 0.
// WTFFFF!!! porqueeEE no debería ser `NaN`??
```

Tabla

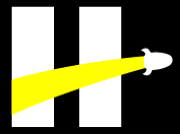


First Class Functions

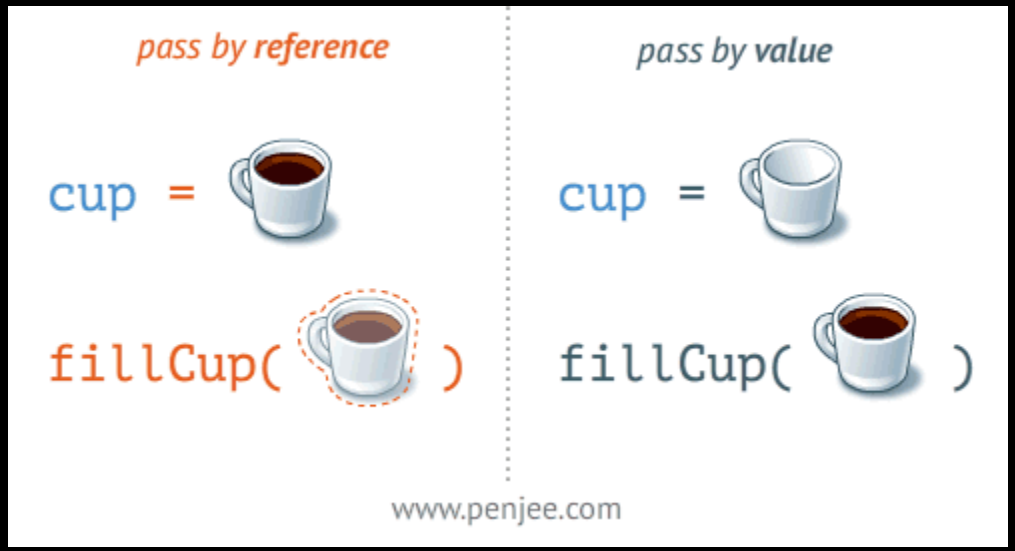


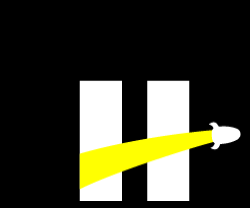
Expresiones y Statements

```
1  // Expresion
2
3  1 + 1;
4  a = 3;
5
6  //Statement
7
8  if (condicion) {
9          // bloque de código
10 }
11
12 // function statement
13
14 function saludo(){
15         console.log('hola');
16 }
17
18 // function expression
19
20 var saludo = function(){
21         console.log('Hola!');
22 }
23
24 console.log(function(){
25         //hola;
26 })
```

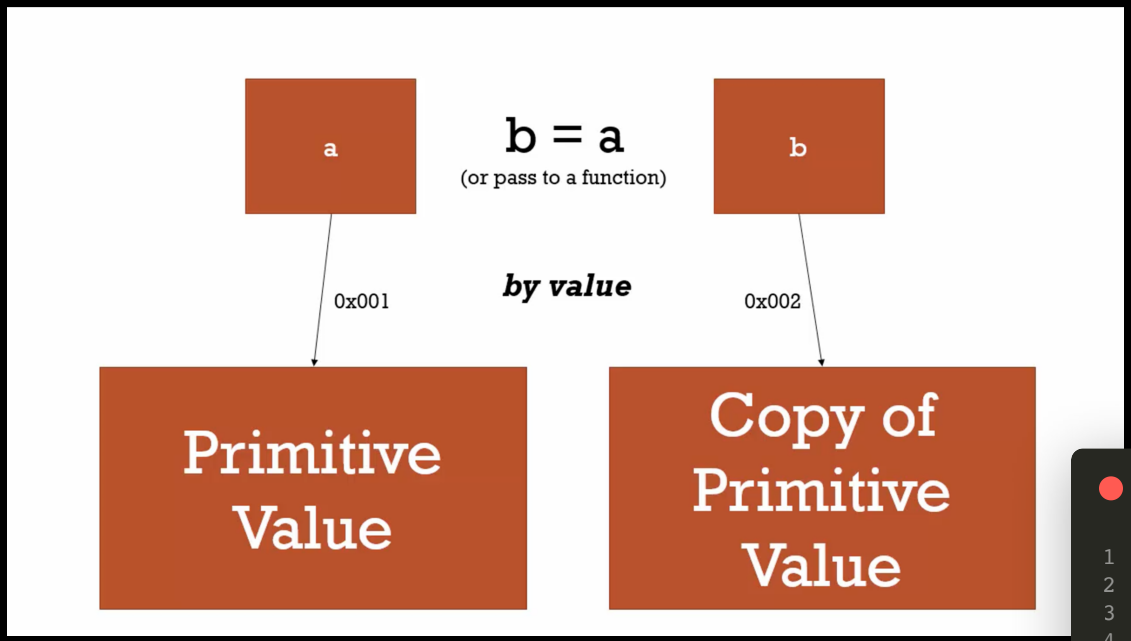


Valor y Referencia



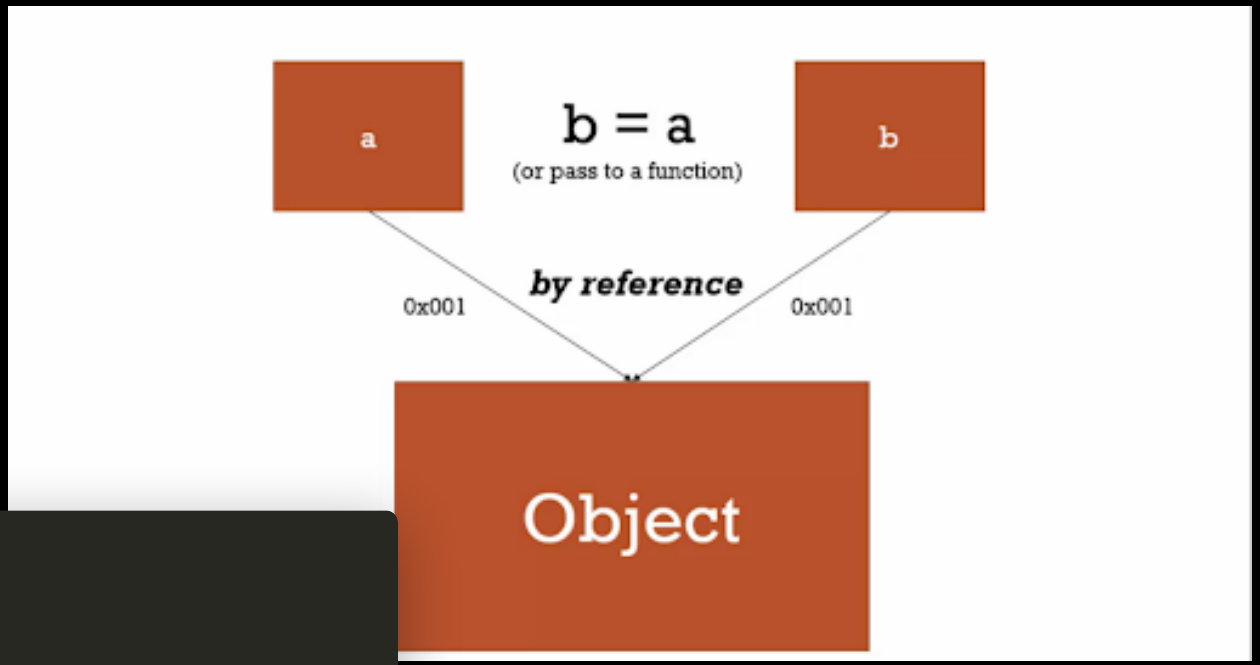


Valor y Referencia

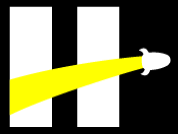



```
1 var a = 1;
2 var b = 2;
3
4 a = b;
5
6 b = 3;
7
8 console.log(a) // 2
9 console.log(b) // 3
```

Valor y Referencia



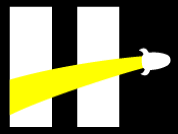

```
1  var a;
2  var b = { nombre : 'hola'};
3
4  a = b ;
5
6  b.nombre = 'Chao';
7
8  console.log(a.nombre); // 'Chao'
9  // Cuando se hizo la asignación se pasó
10 // la referencia de b, por lo tanto
11 // cuando cambiamos la propiedad nombre
12 // de b, se ve reflejado en a
13 // porque ambas variables "apuntan"
14 // al mismo objeto en memoriavar a = 1;
```



This

Contexto global inicial

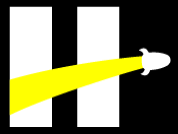
```
1 // En el browser esto es verdad:
2 console.log(this === window); // true
3
4 this.a = 37;
5 console.log(window.a); // 37
```



This

En el contexto de una función

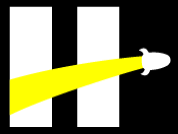
```
1 function f1(){
2   return this;
3 }
4
5 f1() === window; // global object
```



This

Cómo un método de un objeto

```
1  var o = {
2    prop: 37,
3    f: function() {
4      return this.prop;
5    }
6  };
7
8  console.log(o.f()); // logs 37
9  // this hace referencia a `o`
10
11
12 var o = {prop: 37};
13
14 // declaramos la función
15 function loguea() {
16   return this.prop;
17 }
18
19 //agregamos la función como método del objeto `o`
20 o.f = loguea;
21
22 console.log(o.f()); // logs 37
23 // el resultado es le mismo!
```

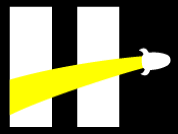


This

Cómo un método de un objeto

```
1  var obj = {
2    nombre: 'Objeto',
3    log   : function(){
4      this.nombre = 'Cambiado'; // this se refiere a este objeto, a `obj`
5      console.log(this)  // obj
6
7      var cambia = function( str ){
8        this.nombre = str;  // Uno esperaria que this sea `obj`
9      }
10
11     cambia('Hoola!!');
12     console.log(this);
13   }
14 }
```

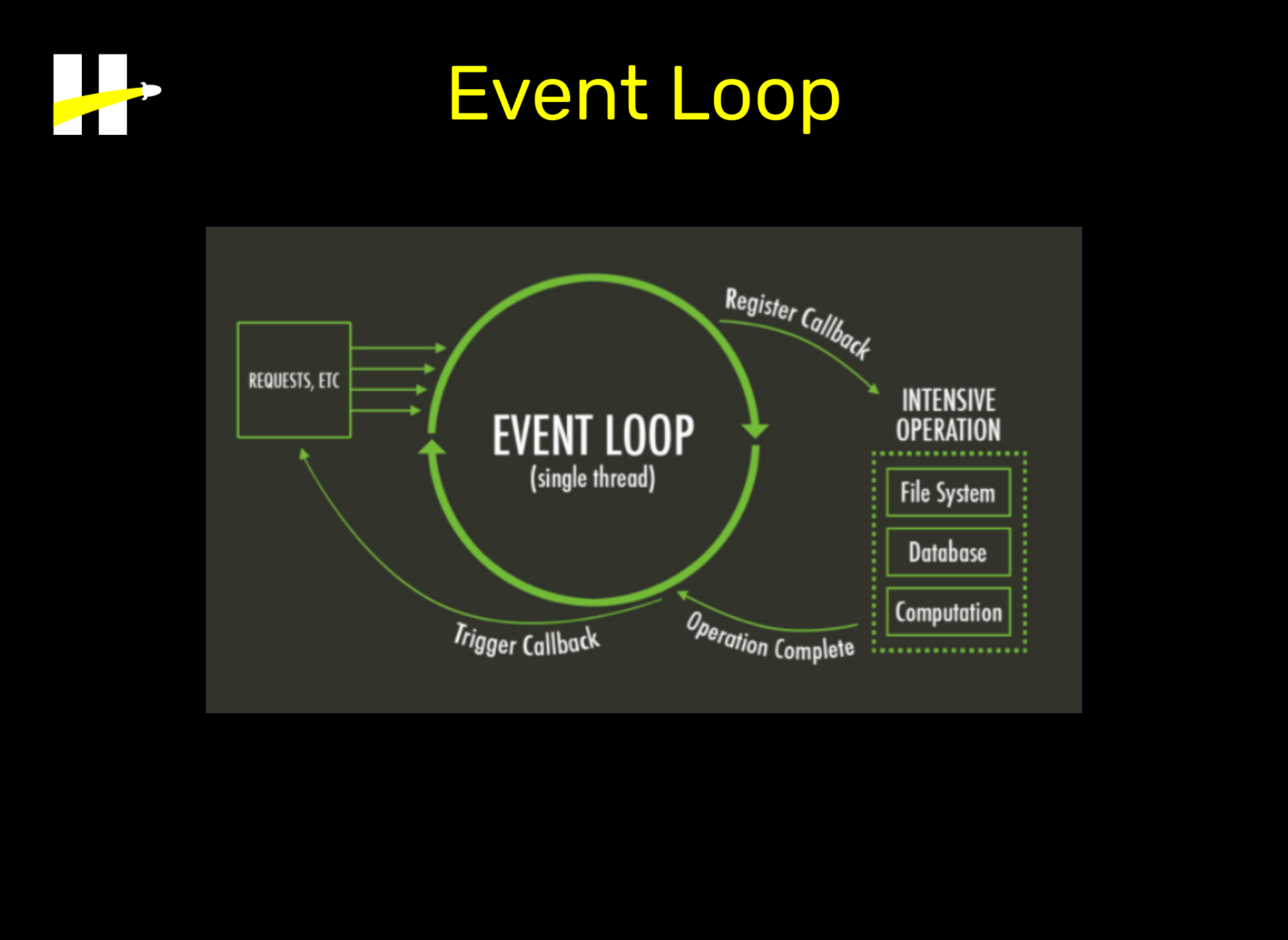
Prácticamente, no podemos saber a ciencia cierta que valor va a tomar el keyword hasta el momento de ejecución de una función. Porque depende fuertemente de cómo haya sido ejecutada.



This

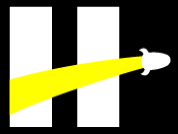
Cómo un método de un objeto

```
1  var obj = {
2    nombre: 'Objeto',
3    log   : function(){
4      this.nombre = 'Cambiado'; // this se refiere a este objeto, a `obj`
5      console.log(this)  // obj
6
7      var that = this; // Guardo la referencia a this
8
9      var cambia = function( str ){
10       that.nombre = str;  // Uso la referencia dentro de esta funcion
11     }
12
13     cambia('Hoola!!');
14     console.log(this);
15   }
16 }
```



Event Loop

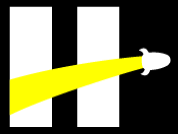
REQUESTS, ETC
EVENT LOOP
(single thread)
Register Callback
INTENSIVE OPERATION
File System
Database
Computation
Trigger Callback
Operation Complete



Event Loop

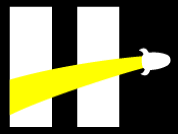
```
1
2
3  function saludarMasTarde(){
4          var saludo = 'Hola';
5
6          setTimeout( function(){
7                  console.log(saludo);
8          },3000)
9  };
10
11 saludarMasTarde();
```

< DEMO />

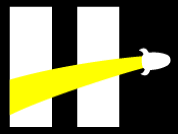


Event Loop

< Ejemplo />

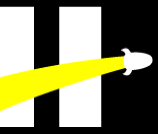


Parte II:
Closures

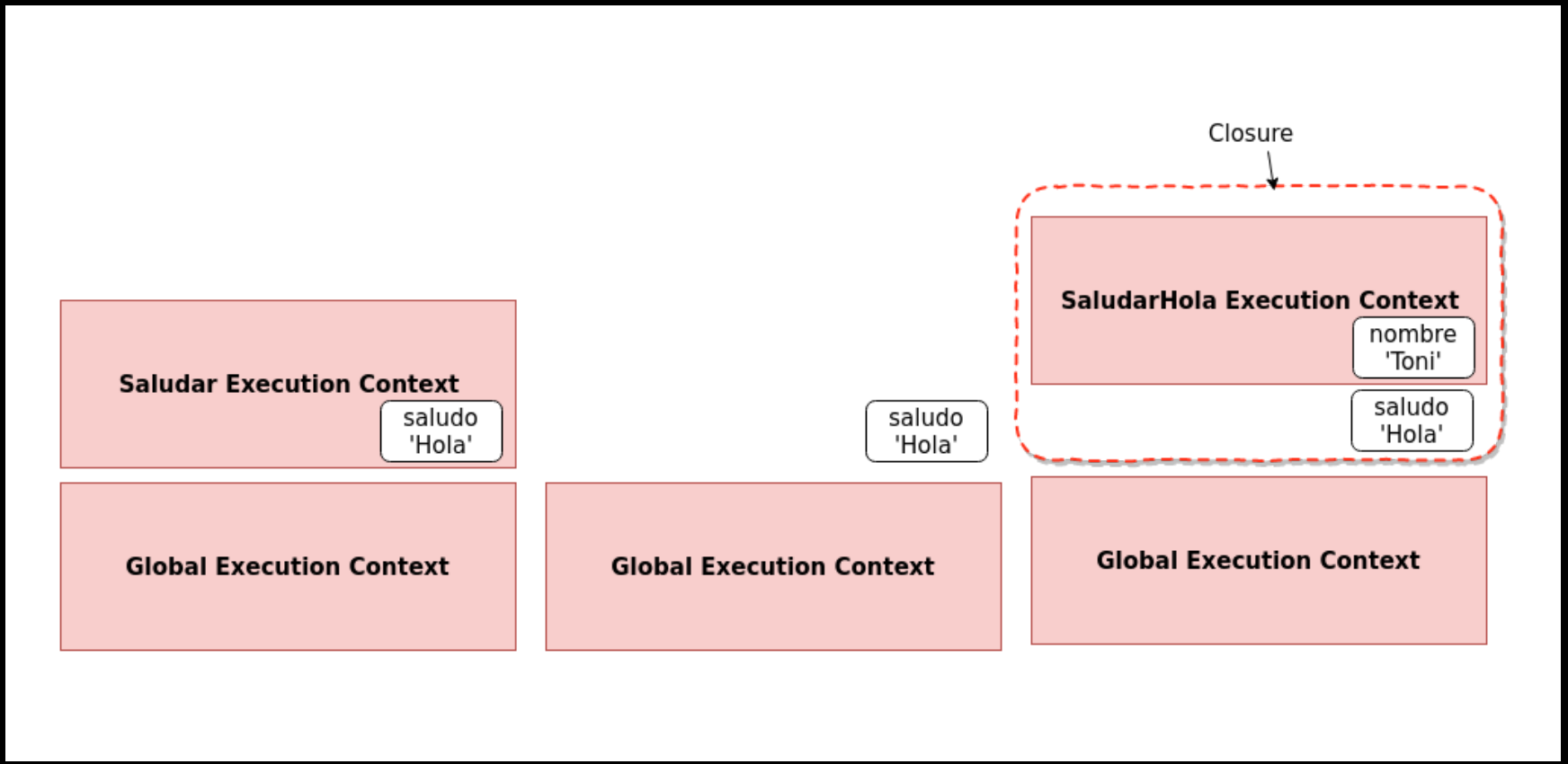


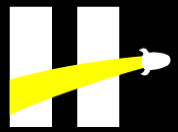
Closures

```
1 function saludar( saludo ){
2         return function( nombre ){
3                 console.log(saludo + ' ' + nombre);
4         }
5 }
6
7 var saludarHola = saludar('Hola'); // Esto devuelve una función
8
9 saludarHola('Toni'); // 'Hola Toni'
```



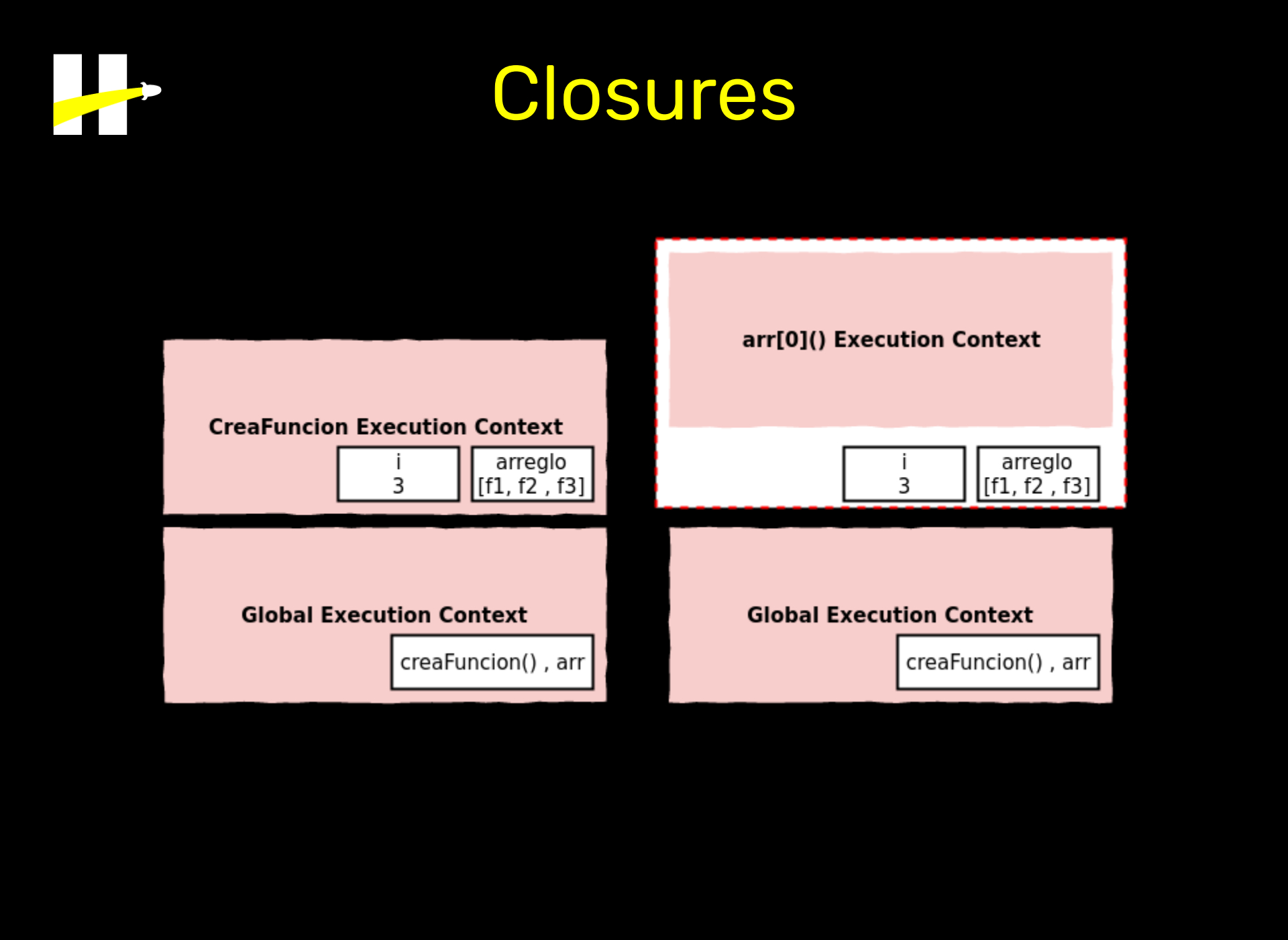
Closures

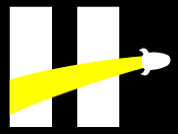

Closure
SaludarHola Execution Context
nombre
'Toni'
saludo
'Hola'
Saludar Execution Context
saludo
'Hola'
saludo
'Hola'
Global Execution Context
Global Execution Context
Global Execution Context



Closures

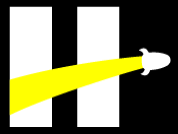
```
1  var creaFuncion = function(){
2          var arreglo = [];
3
4          for ( var i=0; i < 3; i++){
5                  arreglo.push(
6                          function(){
7                                  console.log(i);
8                          }
9                  )
10         }
11         return arreglo;
12 }
13
14 var arr = creaFuncion();
15
16 arr[0]() // 3 sale un 3, qué esperaban ustedes??
17 arr[1]() // 3
18 arr[2]() // 3
```


Closures
CreaFuncion Execution Context
i
3
arreglo
[f1, f2 , f3]
Global Execution Context
creaFuncion() , arr
arr[0]() Execution Context
i
3
arreglo
[f1, f2 , f3]
Global Execution Context
creaFuncion() , arr



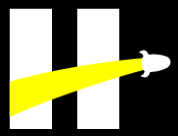
Closures

```
1  var creaFuncion = function(){
2          var arreglo = [];
3          for ( var i=0; i < 3; i++){
4          // IIFE
5                  arreglo.push(
6                          (function(j){
7                                  return function() {console.log(j);}
8                          }(i))
9                  )
10         }
11         return arreglo;
12 }
13
14 var arr = creaFuncion();
15
16 arr[0]() // 1
17 arr[1]() // 2
18 arr[2]() // 3
```

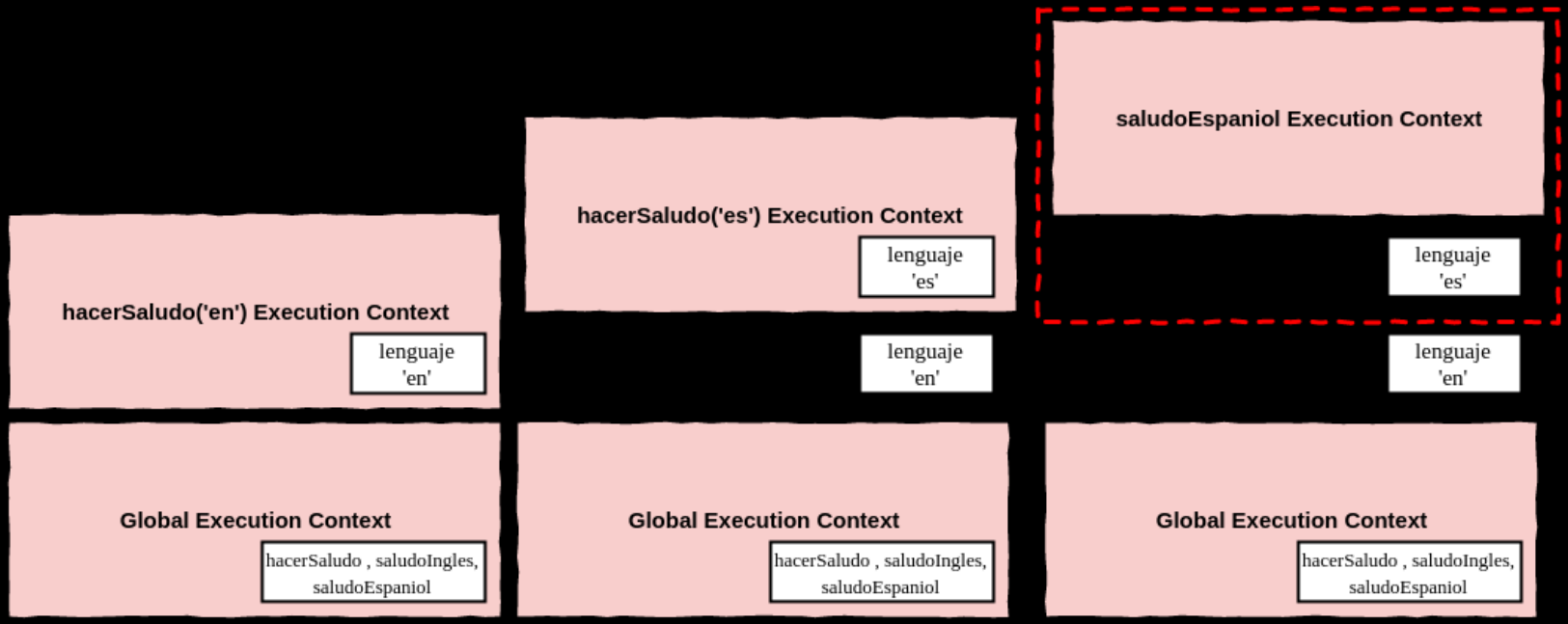


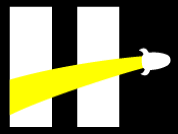
Closures

```
1  function hacerSaludo( lenguaje ){
2          if ( lenguaje === 'en'){
3                  return function(){
4                          console.log('Hi!');
5                  }
6          }
7
8          if ( lenguaje === 'es'){
9                  return function(){
10                         console.log('Hola!');
11                 }
12         }
13 }
14
15 var saludoIngles = hacerSaludo('en');
16 var saludoEspaniol = hacerSaludo('es');
```

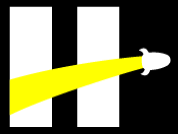


Closures


saludoEspaniol Execution Context
lenguaje
'es'
lenguaje
'en'
hacerSaludo('es') Execution Context
lenguaje
'es'
lenguaje
'en'
hacerSaludo('en') Execution Context
lenguaje
'en'
Global Execution Context
hacerSaludo , saludoIngles,
saludoEspaniol
Global Execution Context
hacerSaludo , saludoIngles,
saludoEspaniol
Global Execution Context
hacerSaludo , saludoIngles,
saludoEspaniol



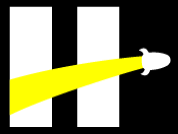
Bind, Call & Apply



Bind, Call & Apply

```
1  var persona = {
2          nombre: 'Guille',
3          apellido: 'Aszyn',
4  }
5
6  var logNombre = function(){
7          console.log(this.nombre);
8  }
9
10 var logNombrePersona = logNombre.bind(persona);
11 // el primer parametro de bind es el this!
12 logNombrePersona();
13
14 // BIND DEVUELVE UNA FUNCION!
```

Cuando vimos el keyword this, dijimos que el interprete era el que manejaba el valor de este. Bueno, esto no es del todo cierto, hay una serie de variables que nos van a permitir poder setear nosotros el keyword this.

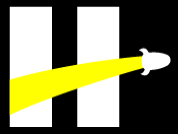


Bind, Call & Apply

```
1 function multiplica(a, b){
2         return a * b;
3 }
4
5 var multiplicaPorDos = multiplica.bind(this, 2);
6 // el Bind le `bindeó` el 2 al argumento a.
7 // y devolvió una función nueva con ese parámetro bindeado.
```

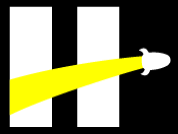
Bind acepta más parámetros, el primer siempre es el `this`, los siguente sirven para bindear parámetros de una función.

Esto se conoce como function currying.



Bind, Call & Apply

```
1  var persona = {
2          nombre: 'Guille',
3          apellido: 'Aszyn',
4  }
5
6  var logNombre = function(){
7          console.log(this.nombre);
8  }
9
10 // el primer parametro de call es el this!
11 logNombre.call(persona);
12
13 // Call hace lo mismo que Bind, solo que invoca la función,
14 // no devuelve una nueva.
15 // tambien bindea argumentos!
16 //
17 var logNombre = function(arg1, arg2){
18         console.log(arg1 +' '+ this.nombre +' '+ arg2);
19 }
20
21 logNombre.call(persona, 'Hola', ', Cómo estas?');
22
23 ////Hola Guille, Cómo estas?
```



Bind, Call & Apply

```
// Apply es igual a call, solo que el segundo argumento es un
// arreglo.

var logNombre = function(arg1, arg2){
        console.log(arg1 +' '+ this.nombre +' '+ arg2);
}

logNombre.apply(persona, ['Hola', ', Cómo estas?']);
//Hola Guille , Cómo estas?
```