# Web Fundamentals

Session-02

Javascript Scope, Types, Closure

Thanos is on a mission to make his website standout from his rest of universe

# Web Fundamentals

Session-0
2

JS Fundamentals

# Agenda : Javascript Basics

01 JS Types ( Objects, Strings, Arrays)

02 Value Vs Reference Types

03 IIFE, Closures

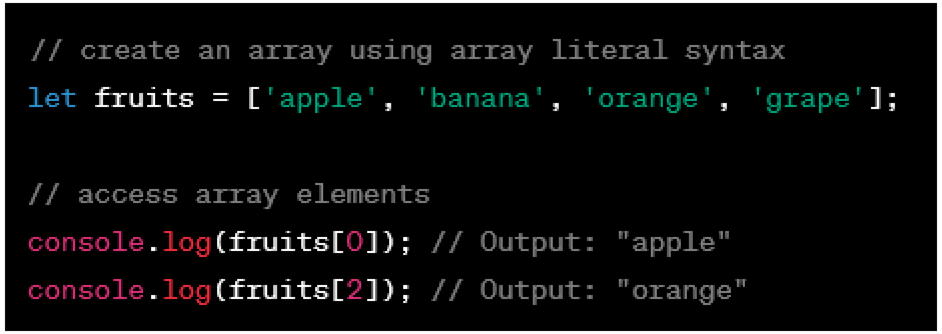
04 Strict Mode, this keyword

05 Call/Bind/Apply

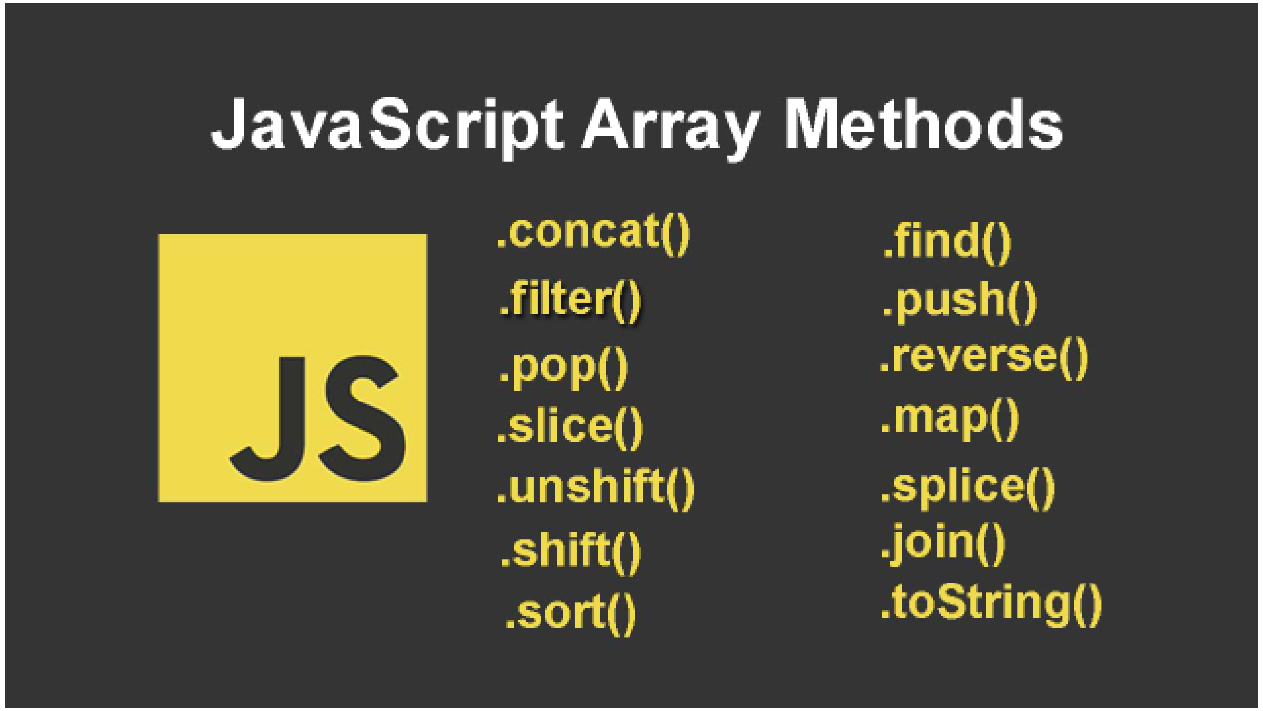
06 Hands-On

# JS Array

Array is an ordered collection of elements, which can be of any data type



```
// create an array using array literal syntax
let fruits = ['apple', 'banana', 'orange', 'grape'];

// access array elements
console.log(fruits[0]); // Output: "apple"
console.log(fruits[2]); // Output: "orange"
```



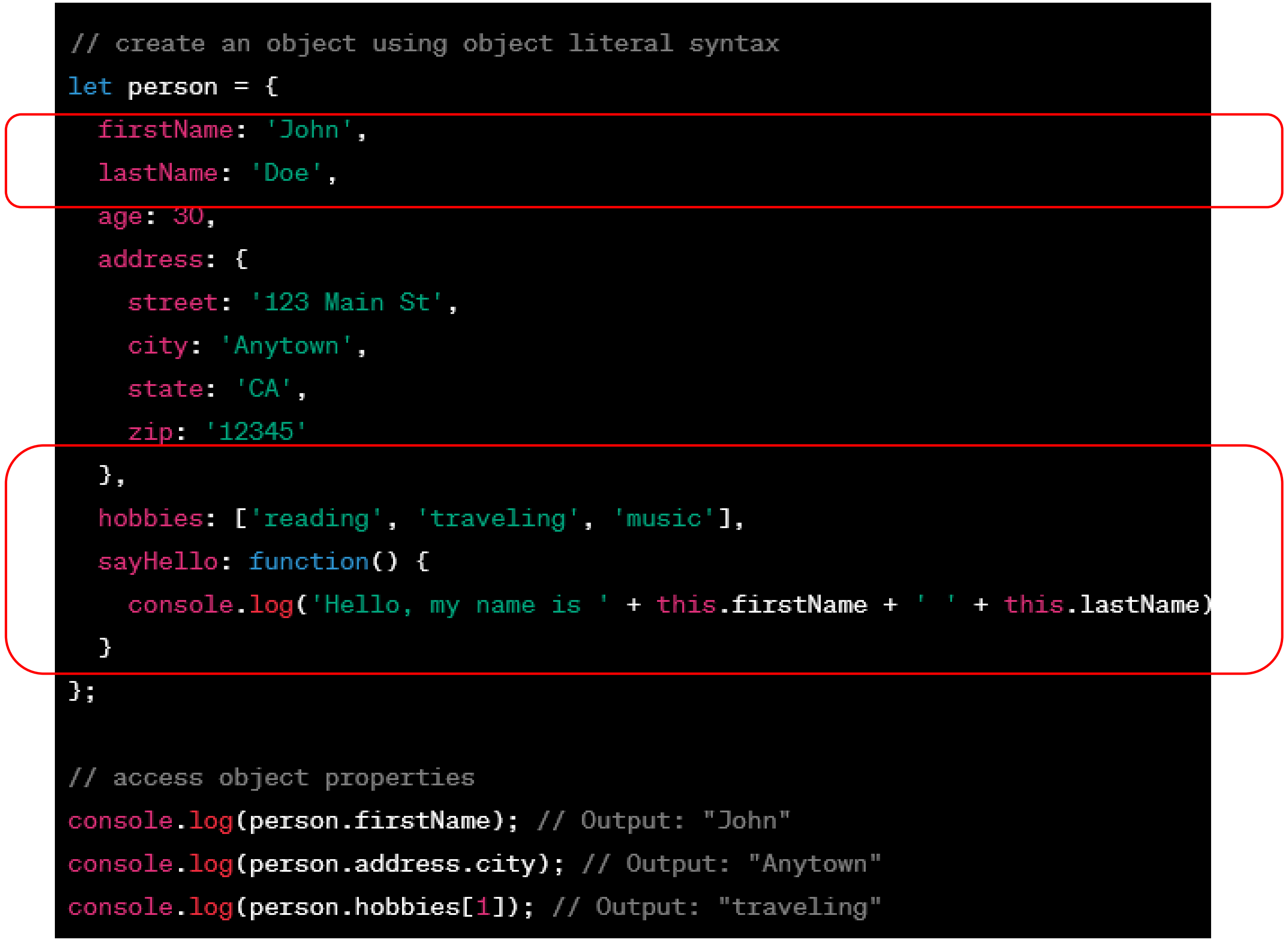
JavaScript Array Methods

.concat()
.filter()
.pop()
.slice()
.unshift()
.shift()
.sort()

.find()
.push()
.reverse()
.map()
.splice()
.join()
.toString()

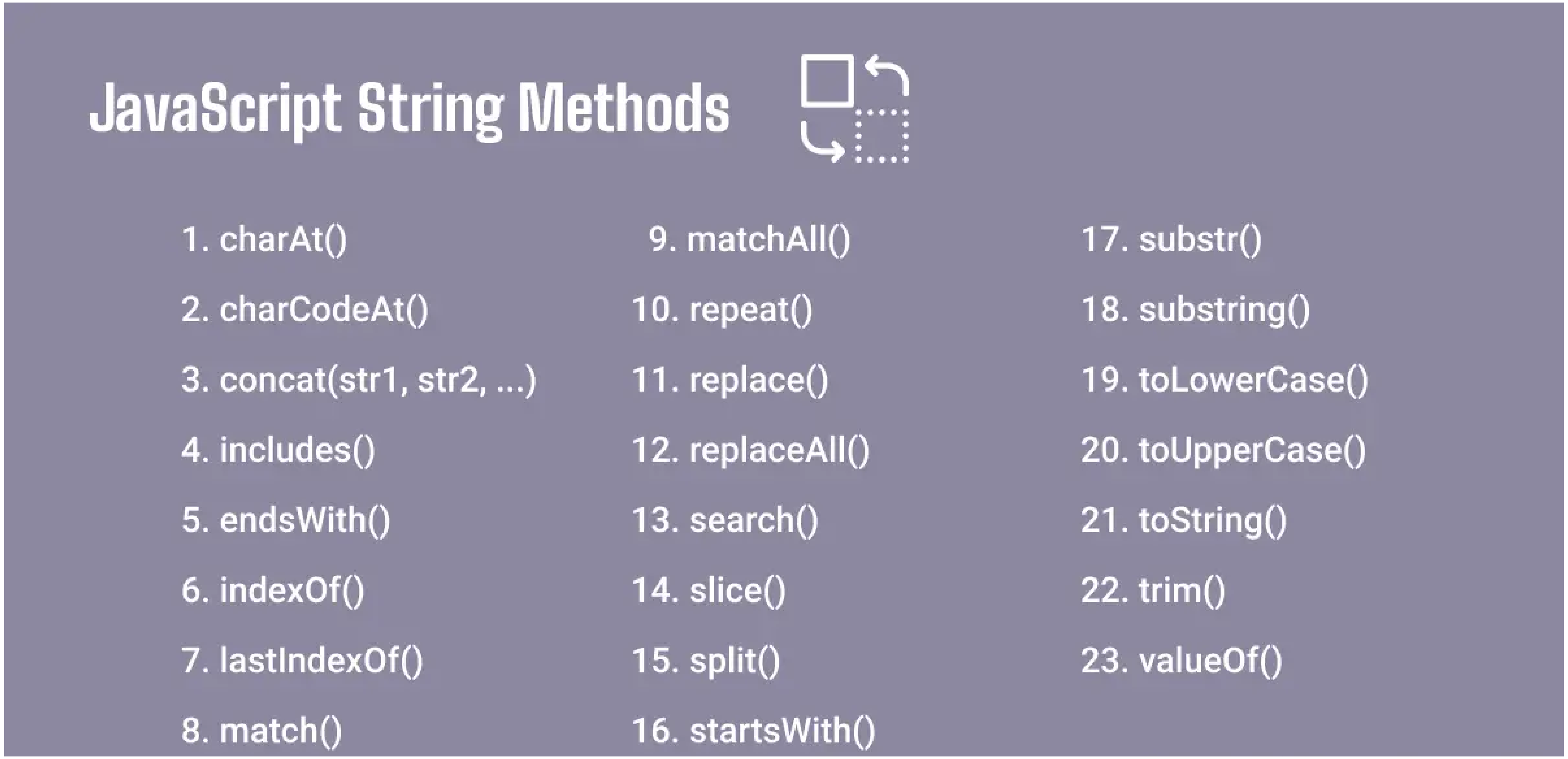
# JS Objects

- Object is a **collection of properties** that have a name and a value
- Objects can contain **other objects, functions, and even arrays**, making them a powerful way to store and organize data



```
// create an object using object literal syntax
let person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30,
  address: {
    street: '123 Main St',
    city: 'Anytown',
    state: 'CA',
    zip: '12345'
  },
  hobbies: ['reading', 'traveling', 'music'],
  sayHello: function() {
    console.log('Hello, my name is ' + this.firstName + ' ' + this.lastName)
  }
};

// access object properties
console.log(person.firstName); // Output: "John"
console.log(person.address.city); // Output: "Anytown"
console.log(person.hobbies[1]); // Output: "traveling"
```

# String Object & Methods

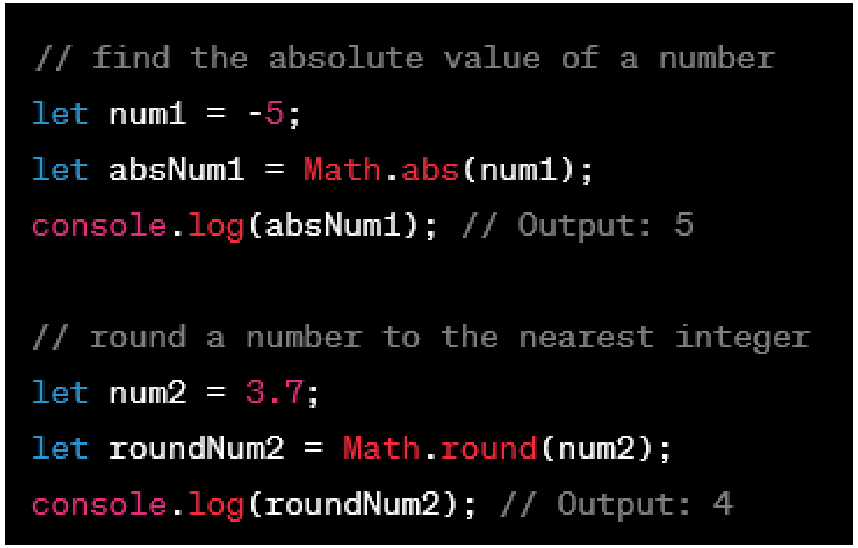


## JavaScript String Methods

1. charAt()
2. charCodeAt()
3. concat(str1, str2, ...)
4. includes()
5. endsWith()
6. indexOf()
7. lastIndexOf()
8. match()
9. matchAll()
10. repeat()
11. replace()
12. replaceAll()
13. search()
14. slice()
15. split()
16. startsWith()
17. substr()
18. substring()
19. toLowerCase()
20. toUpperCase()
21. toString()
22. trim()
23. valueOf()

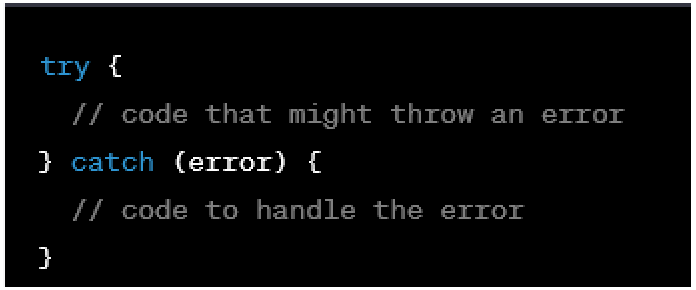
# Math Object

Math object in JavaScript provides a set of built-in mathematical functions and constants



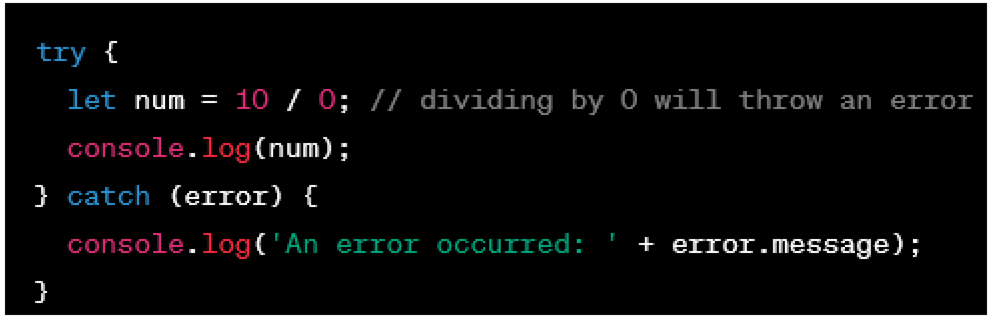
```
// find the absolute value of a number
let num1 = -5;
let absNum1 = Math.abs(num1);
console.log(absNum1); // Output: 5

// round a number to the nearest integer
let num2 = 3.7;
let roundNum2 = Math.round(num2);
console.log(roundNum2); // Output: 4
```

# Exception Handling



```javascript
try {
  // code that might throw an error
} catch (error) {
  // code to handle the error
}
```

## Example

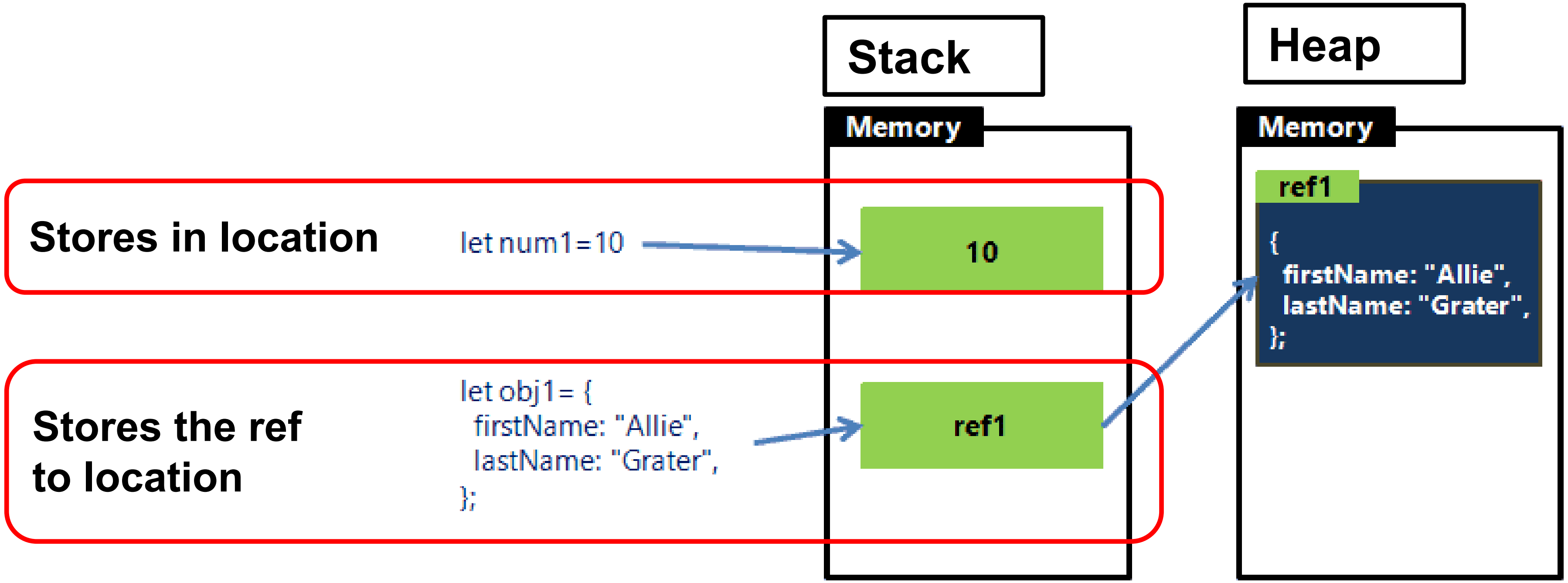


```javascript
try {
  let num = 10 / 0; // dividing by 0 will throw an error
  console.log(num);
} catch (error) {
  console.log('An error occurred: ' + error.message);
}
```

# <Value type (Primitive) & Reference type (Non-Primitive)/>

What is a Value type and Reference type?

Primary Difference – **How they stored in memory**



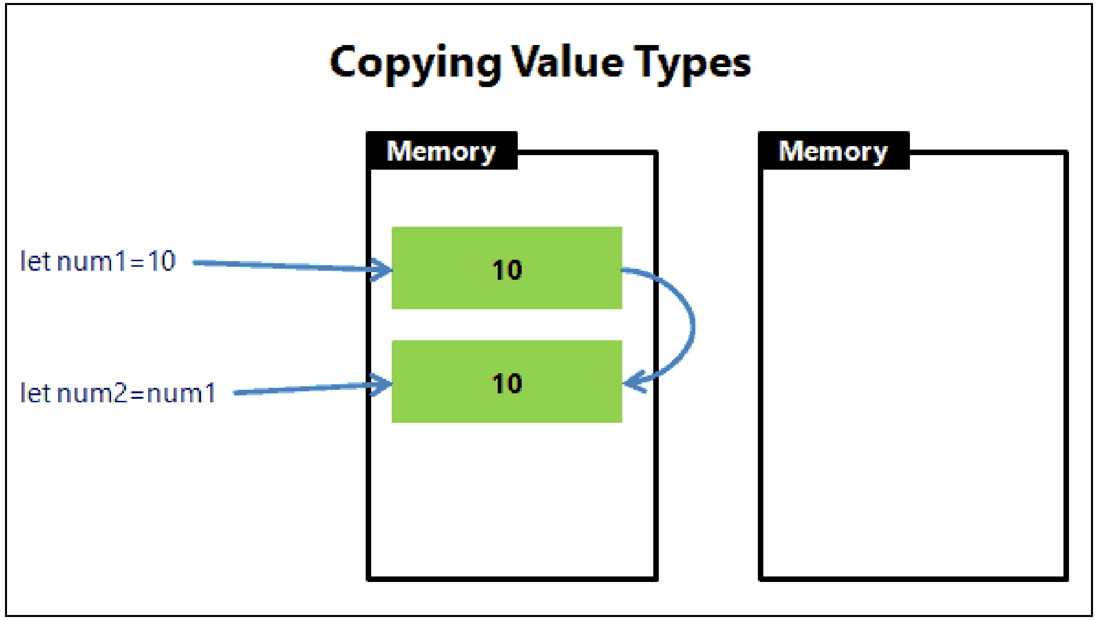
**Stores in location**

```
let num1=10
```

**Stores the ref to location**

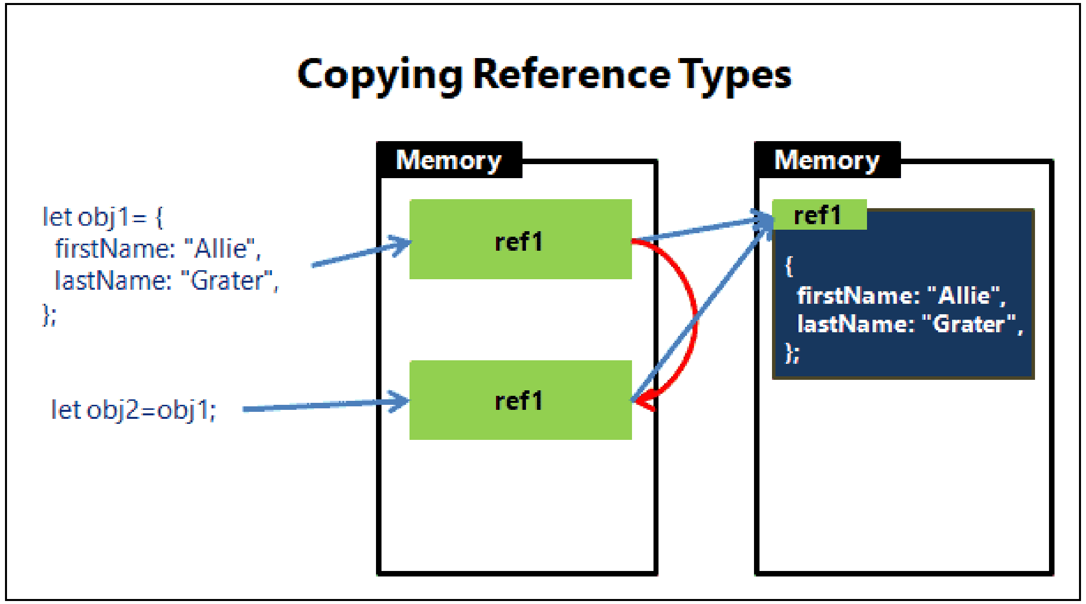
```
let obj1= {
  firstName: "Allie",
  lastName: "Grater",
};
```

# **Copying** Value type (Primitive) & Reference type (Non-Primitive)



## Copying Value Types

```
let num1=10
let num2=num1
```



## Copying Reference Types

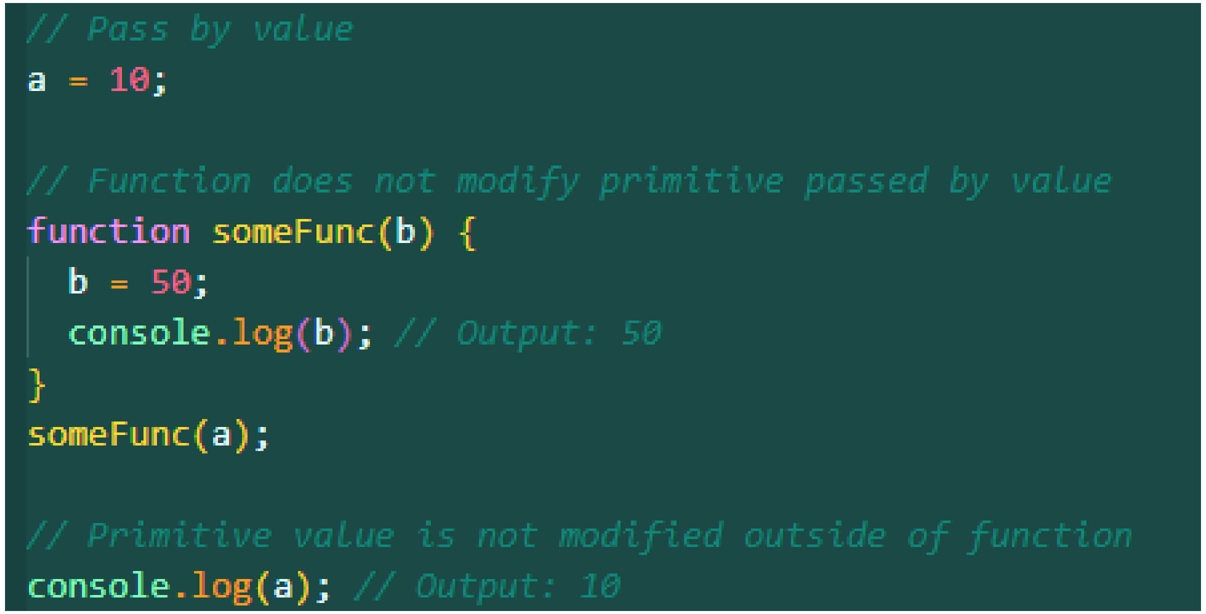
```
let obj1= {
  firstName: "Allie",
  lastName: "Grater",
};
```

```
let obj2=obj1;
```

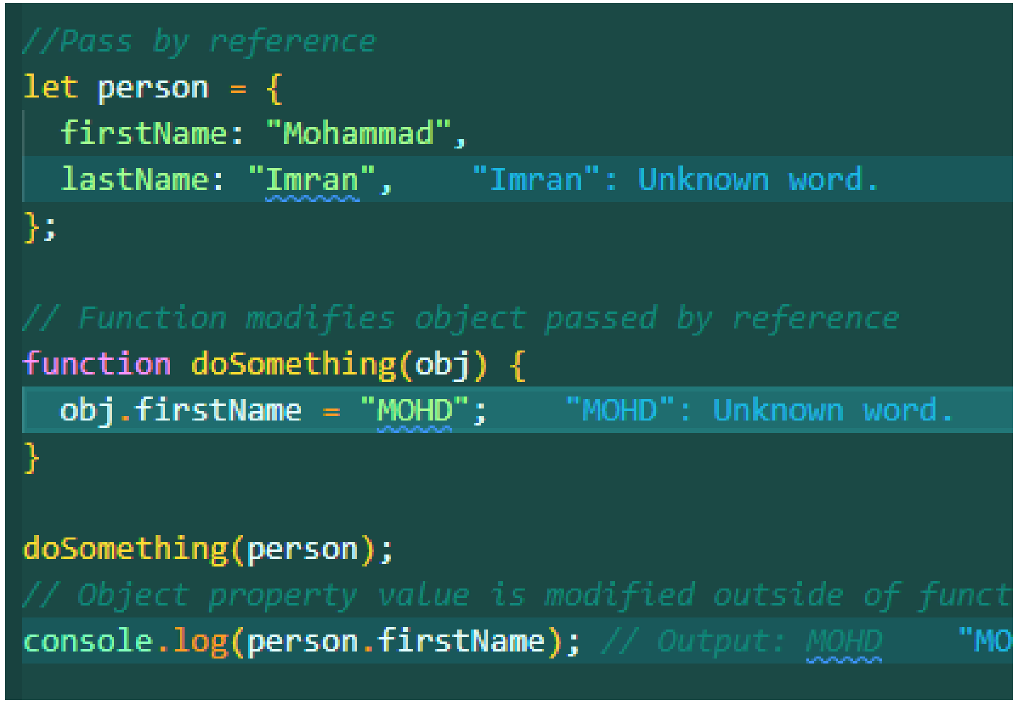
```
{
  firstName: "Allie",
  lastName: "Grater",
};
```

## <Pass by value/>

## <Pass by reference/>



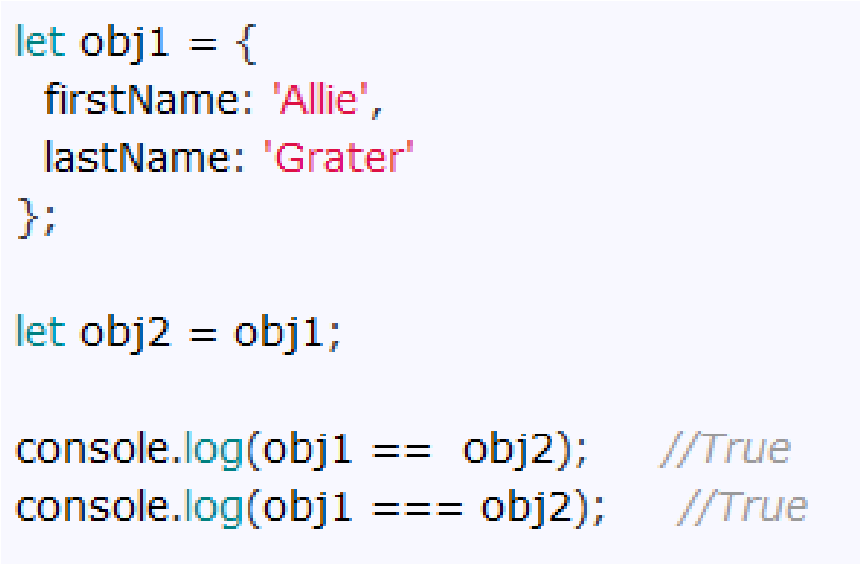
```
// Pass by value
a = 10;

// Function does not modify primitive passed by value
function someFunc(b) {
  b = 50;
  console.log(b); // Output: 50
}
someFunc(a);

// Primitive value is not modified outside of function
console.log(a); // Output: 10
```



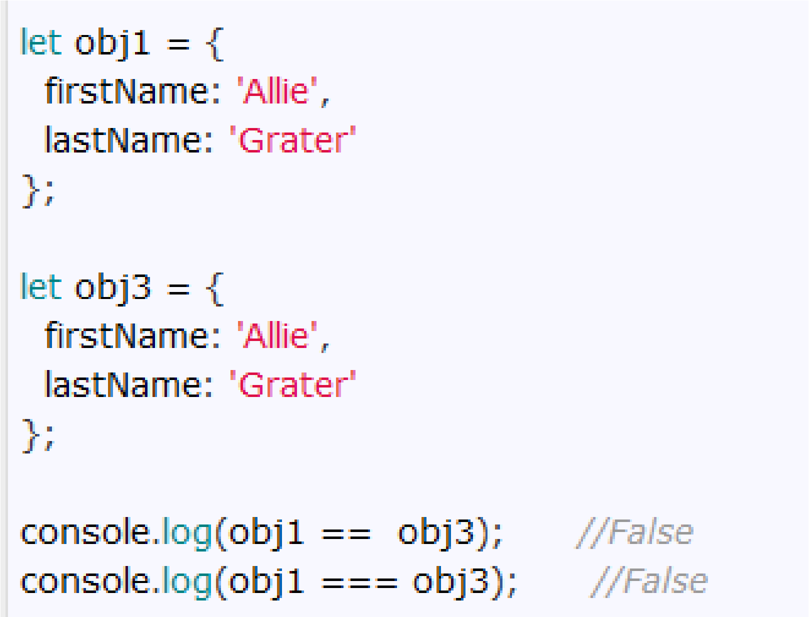
```
//Pass by reference
let person = {
  firstName: "Mohammad",
  lastName: "Imran",    "Imran": Unknown word.
};

// Function modifies object passed by reference
function doSomething(obj) {
  obj.firstName = "MOHD";    "MOHD": Unknown word.
}

doSomething(person);
// Object property value is modified outside of funct
console.log(person.firstName); // Output: MOHD    "MO
```

# Equality Check of reference types

Explain the output, Why?



```
let obj1 = {
  firstName: 'Allie',
  lastName: 'Grater'
};

let obj2 = obj1;

console.log(obj1 ==  obj2);    //True
console.log(obj1 === obj2);    //True
```

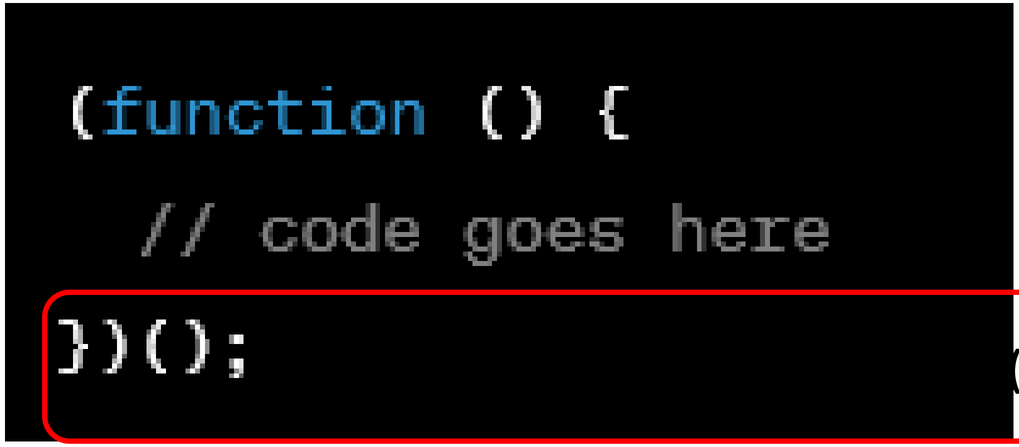


```
let obj1 = {
  firstName: 'Allie',
  lastName: 'Grater'
};

let obj3 = {
  firstName: 'Allie',
  lastName: 'Grater'
};

console.log(obj1 ==  obj3);    //False
console.log(obj1 === obj3);    //False
```

**obj1 & obj2** contains the same reference & value

**obj1 & obj3** contains the different reference though they represent same data

# IIFE (Immediately-invoked Function Expressions)



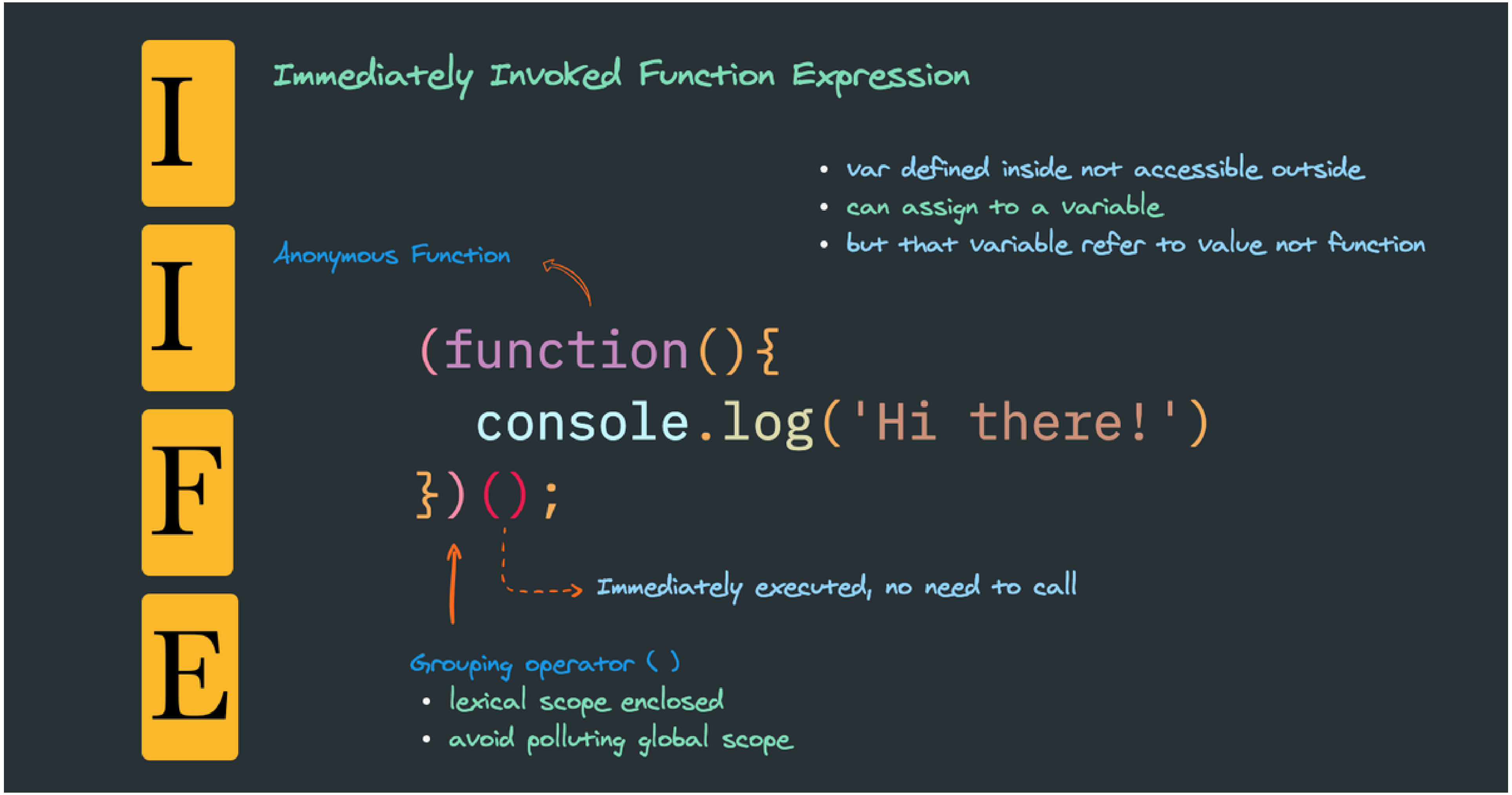
```
(function () {
  // code goes here
})();
```

( )” immediately invokes the function after it is defined.

Why do we need IIFE?

- Primary uses of an IIFE is to create a **private scope for variables**
- **Preventing them from interfering with other code.**

# IIFE (Immediately-invoked Function Expressions)



Immediately Invoked Function Expression

- var defined inside not accessible outside
- can assign to a variable
- but that variable refer to value not function

Anonymous Function

```
(function(){
  console.log('Hi there!')
})();
```

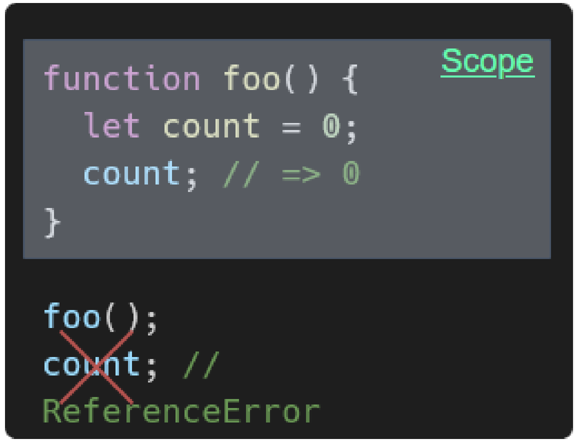
Immediately executed, no need to call

Grouping operator ( )

- lexical scope enclosed
- avoid polluting global scope

# Why <Closure/>?

**Problem with Scope:** Outer functions cannot access inner functions variables



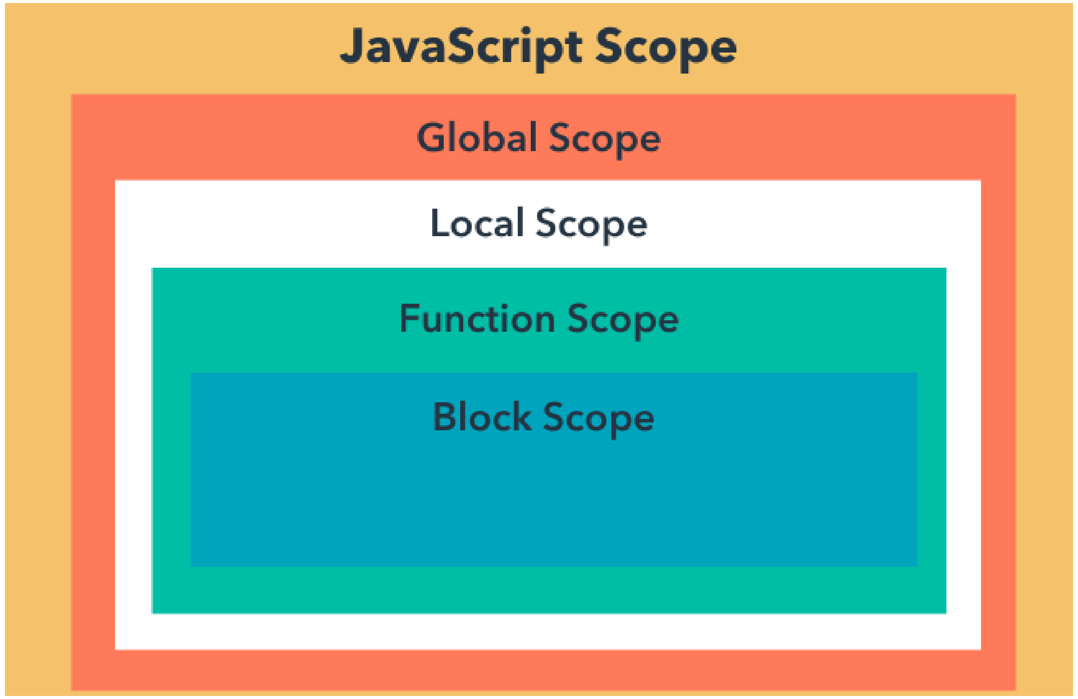
Scope

```
function foo() {
  let count = 0;
  count; // => 0
}

foo();
count; //
ReferenceError
```

To understand better

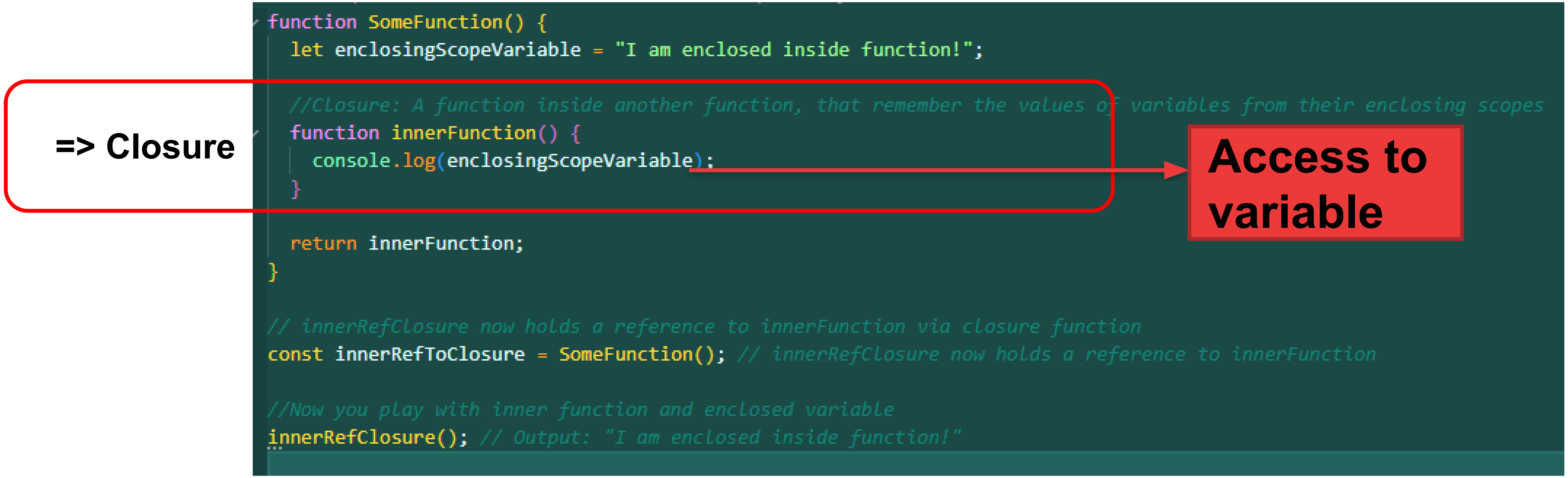
## Let's **Recap Scope** once

Scope refers to the **visibility and accessibility of variables and functions** in different parts of your code.



# <Closures/>

Closures allow us to **create functions that remember the values of variables** from their **enclosing scopes**



```
function SomeFunction() {
  let enclosingScopeVariable = "I am enclosed inside function!";

  //Closure: A function inside another function, that remember the values of variables from their enclosing scopes
  function innerFunction() {
    console.log(enclosingScopeVariable);
  }

  return innerFunction;
}

// innerRefClosure now holds a reference to innerFunction via closure function
const innerRefToClosure = SomeFunction(); // innerRefClosure now holds a reference to innerFunction

//Now you play with inner function and enclosed variable
innerRefClosure(); // Output: "I am enclosed inside function!"
```

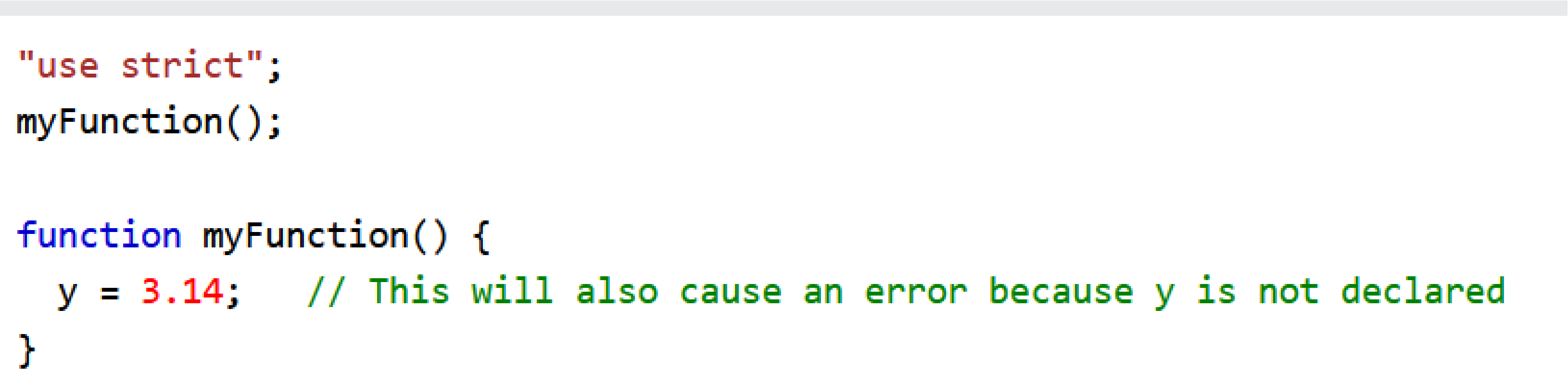
=> Closure

Access to variable

# <Strict Mode/>

What is “strict  mode”?

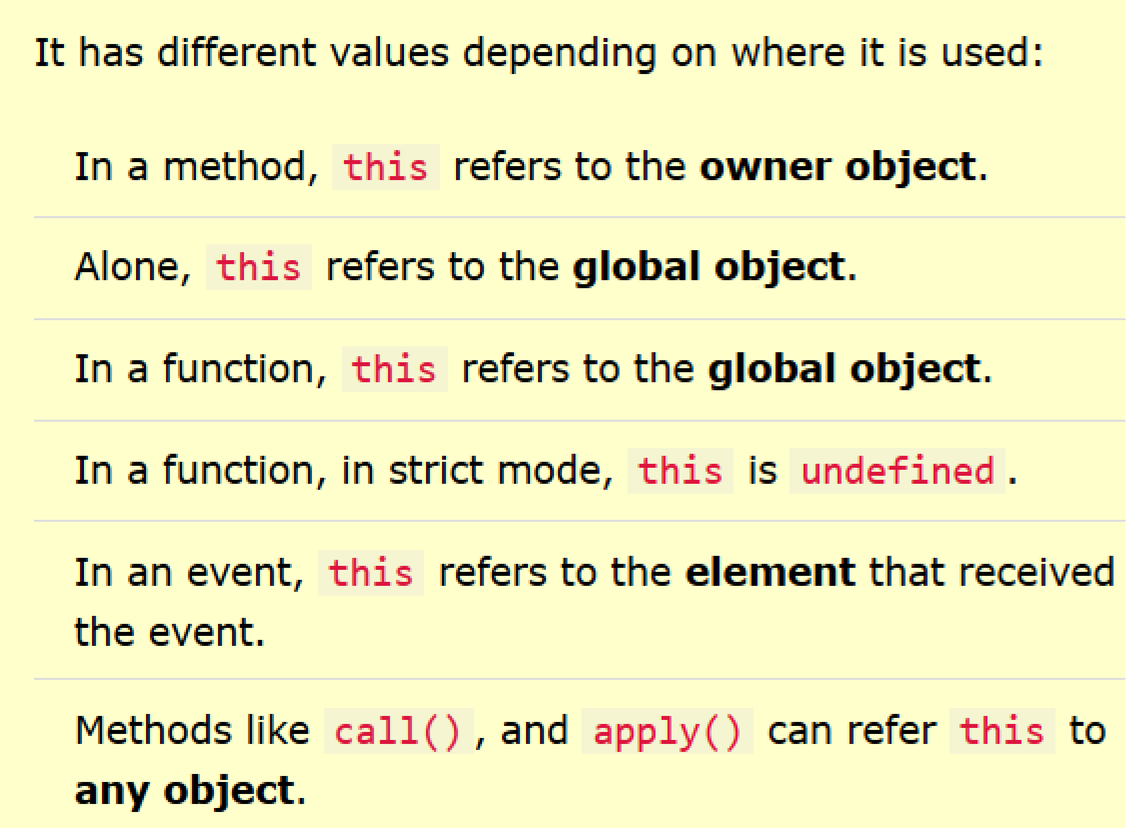
- Throws errors for the things that are **disallowed or restricted** in order to avoid common mistakes and **promote more secure and reliable code**.
- Prohibits some syntax that is likely to be defined in future Javascript



```
"use strict";
myFunction();

function myFunction() {
  y = 3.14;   // This will also cause an error because y is not declared
}
```

# ‘this’- Keyword- Dynamic Context

- The value of **"this" in JavaScript is dynamic** and depends on the context
- “this” - **Refers to the object** that it belongs to
- **Which object depends on how this is being invoked (used or called).**



It has different values depending on where it is used:

In a method, `this` refers to the **owner object**.

Alone, `this` refers to the **global object**.

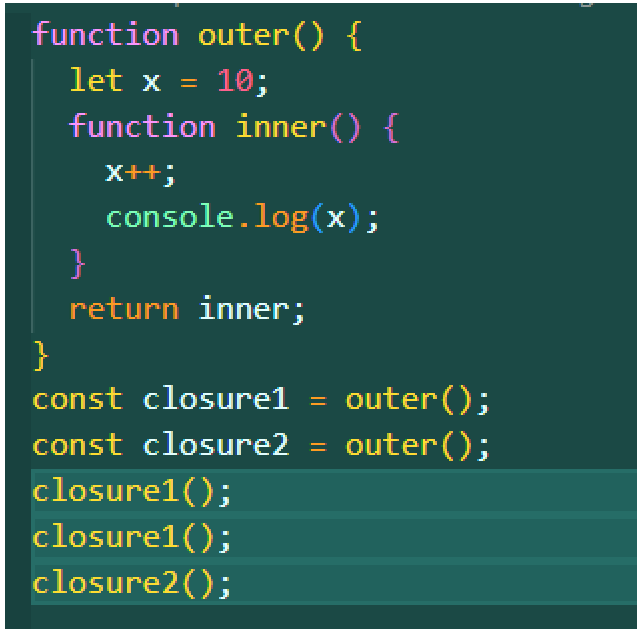
In a function, `this` refers to the **global object**.

In a function, in strict mode, `this` is `undefined`.

In an event, `this` refers to the **element** that received the event.

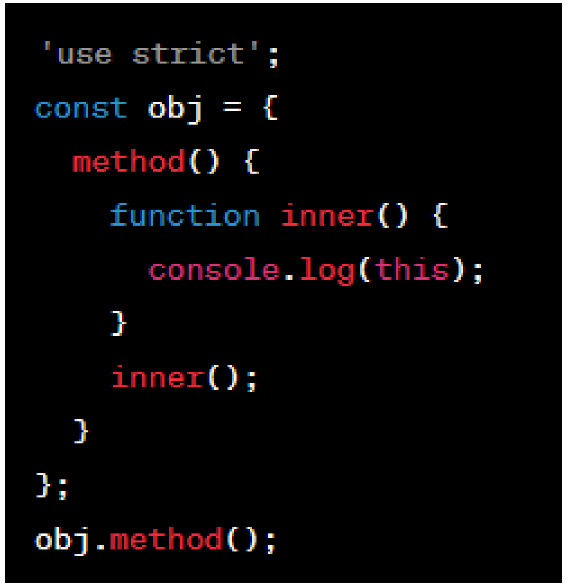
Methods like `call()`, and `apply()` can refer `this` to **any object**.

# Knowledge check: What is the output?



```javascript
function outer() {
  let x = 10;
  function inner() {
    x++;
    console.log(x);
  }
  return inner;
}
const closure1 = outer();
const closure2 = outer();
closure1();
closure1();
closure2();
```

```
11
12
11
```

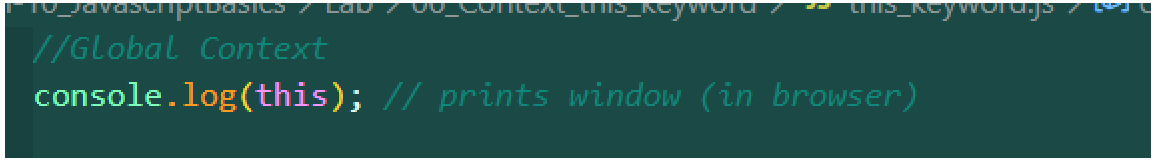


```javascript
'use strict';
const obj = {
  method() {
    function inner() {
      console.log(this);
    }
    inner();
  }
};
obj.method();
```

*Undefined*
because **in strict mode**, the value of this is **not set by default to the global object in a function**. Instead, it remains undefined

# Global Context and "this"

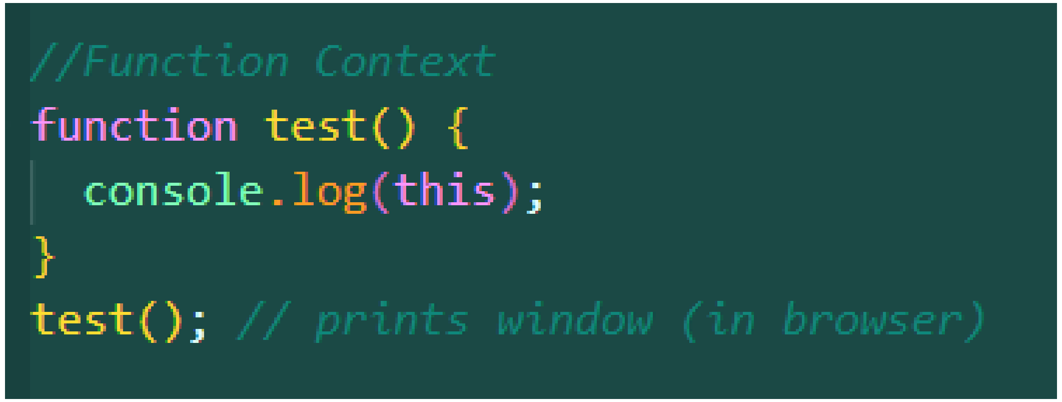
In the global context, the value of "this" is the global object


-10_JavascriptBasics > Lab > 06_Context_this_keyword > JS this_keyword.js >

```
//Global Context
console.log(this); // prints window (in browser)
```

# Function Context and "this"

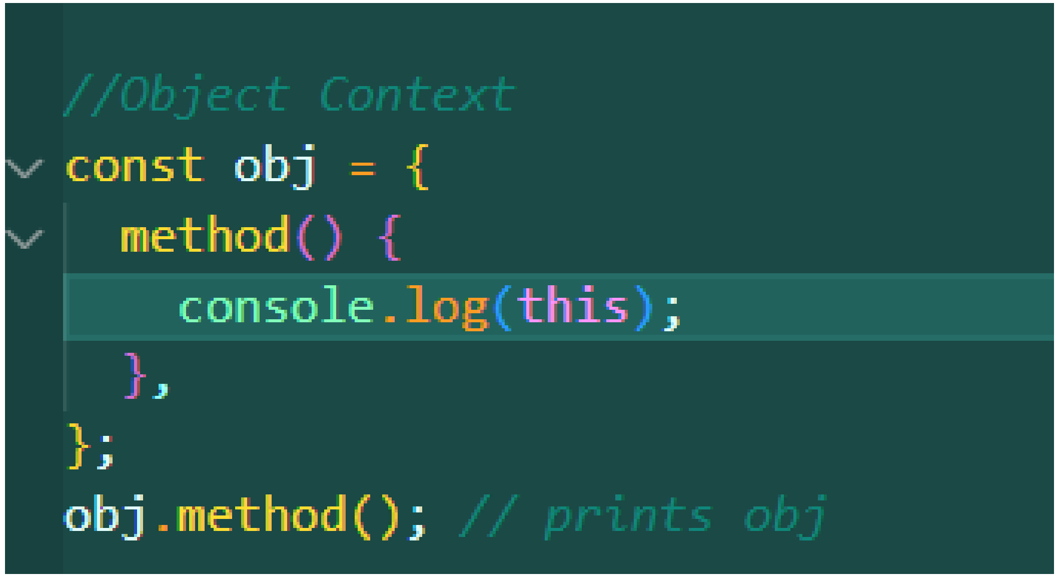
In a function, the value of "this" depends on how the function is called.



```
//Function Context
function test() {
  console.log(this);
}
test(); // prints window (in browser)
```

# Object Method Context and "this"

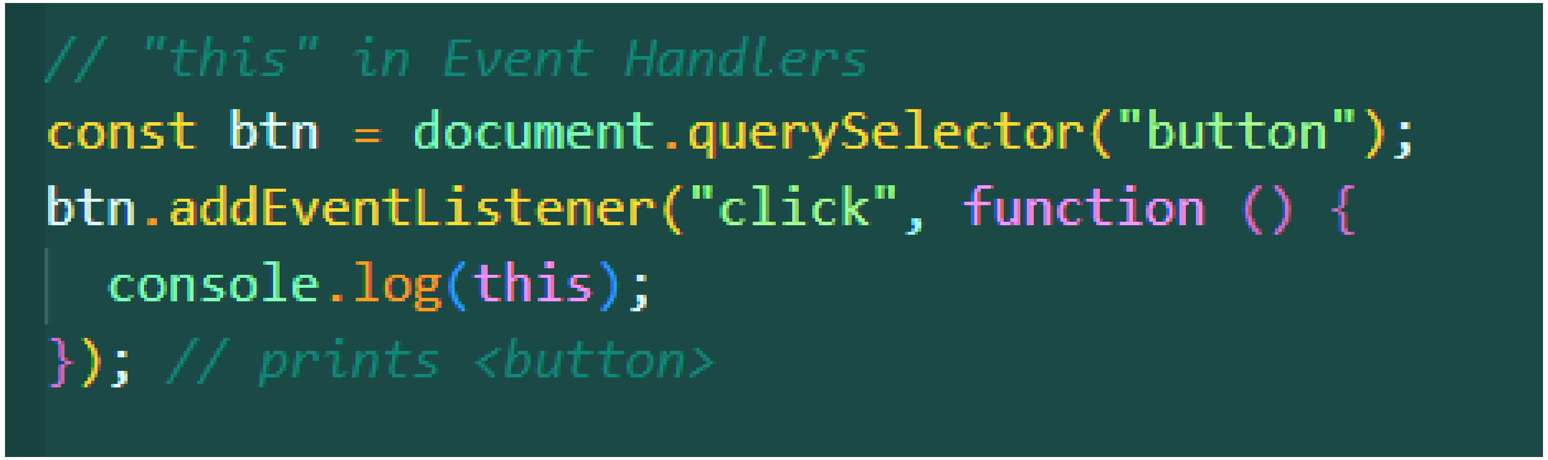
In an object method, the value of "this" refers to the object that the method is a property of.



```
//Object Context
const obj = {
  method() {
    console.log(this);
  },
};
obj.method(); // prints obj
```

# "this" in Event Handlers

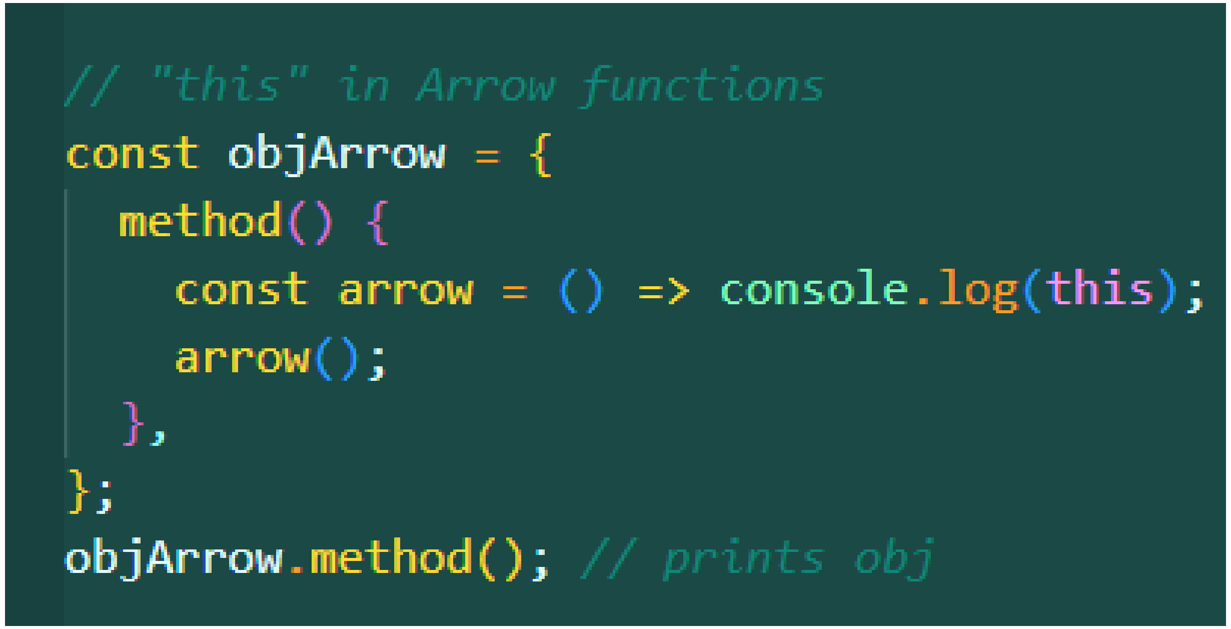
In an object method, the value of "this" refers to the object that the method is a property of.



```
// "this" in Event Handlers
const btn = document.querySelector("button");
btn.addEventListener("click", function () {
  console.log(this);
}); // prints <button>
```

# "this" in Arrow Functions

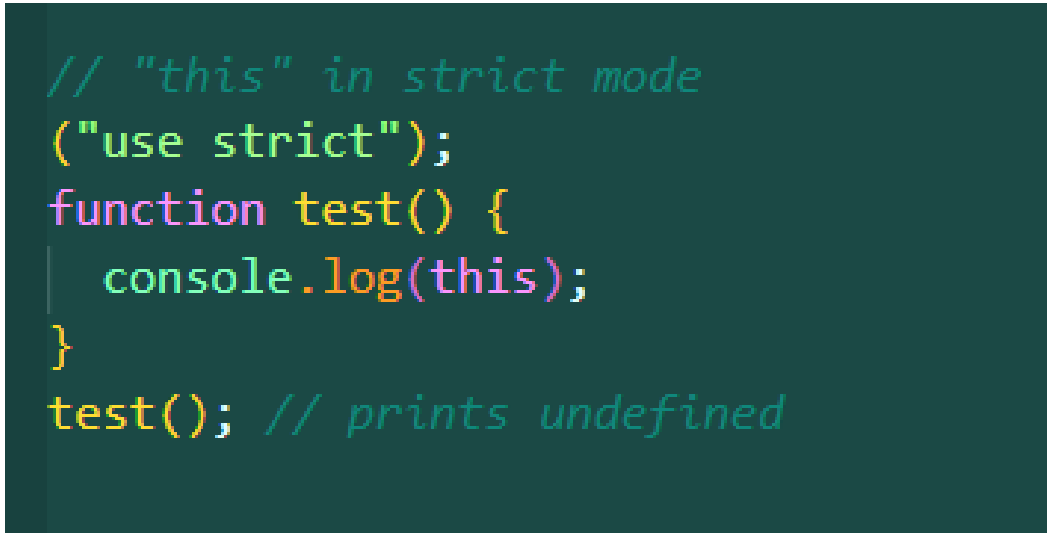
In an object method, the value of "this" refers to the object that the method is a property of.



```
// "this" in Arrow functions
const objArrow = {
  method() {
    const arrow = () => console.log(this);
    arrow();
  },
};
objArrow.method(); // prints obj
```

# "this" in Strict Mode

In an object method, the value of "this" refers to the object that the method is a property of.

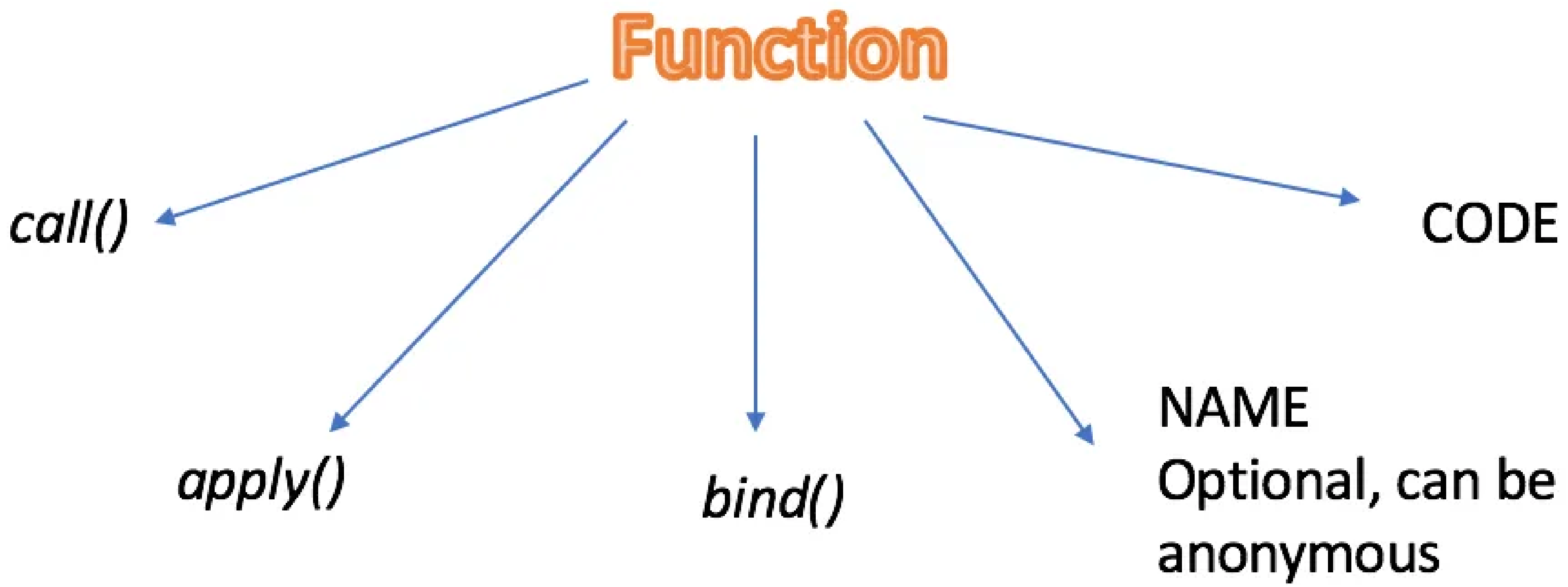


```
// "this" in strict mode
("use strict");
function test() {
  console.log(this);
}
test(); // prints undefined
```

# Functions- Deep Dive

In JS- Every function is Object and Function can be executed multiple ways

**JS function prototype provides out of box methods call(), bind() and apply()**



Function

*call()*

*apply()*

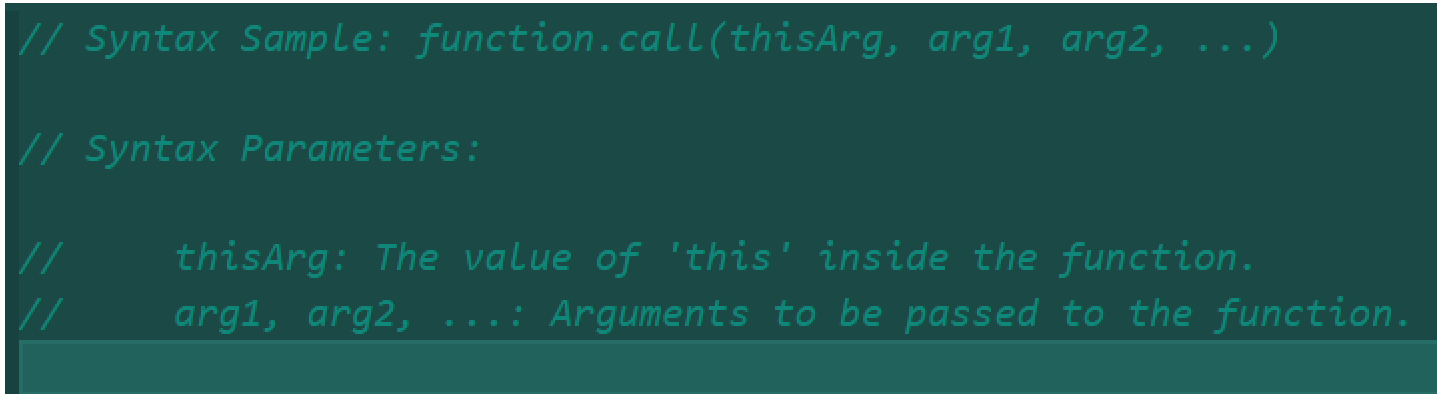
*bind()*

NAME
Optional, can be anonymous

CODE

# Call ( )

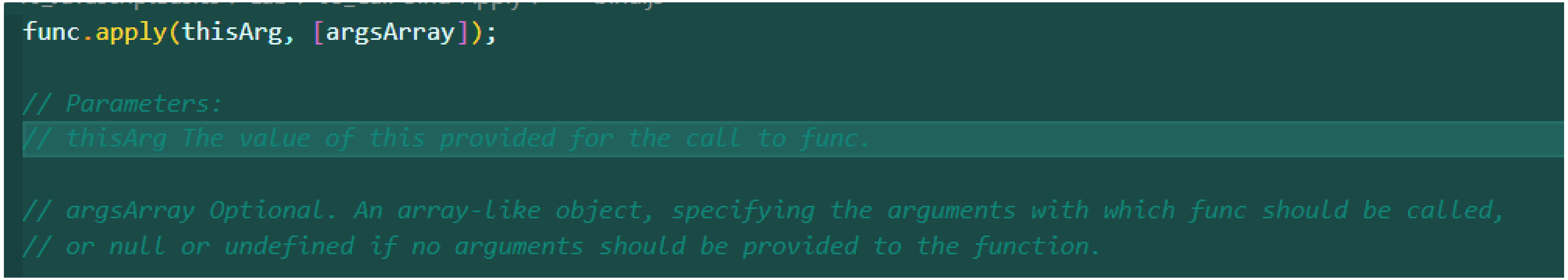
**Call()** a function with a specified 'this' value and arguments provided individually.



```
// Syntax Sample: function.call(thisArg, arg1, arg2, ...)

// Syntax Parameters:

//     thisArg: The value of 'this' inside the function.
//     arg1, arg2, ...: Arguments to be passed to the function.
```

# Apply ( )

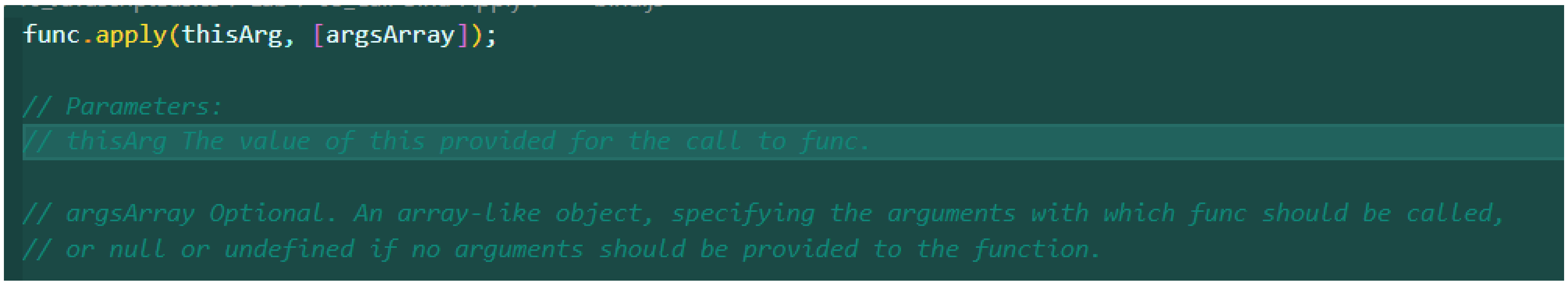
The **apply()** method calls a function with a given this value, and arguments provided as an array



```
func.apply(thisArg, [argsArray]);

// Parameters:
// thisArg The value of this provided for the call to func.

// argsArray Optional. An array-like object, specifying the arguments with which func should be called,
// or null or undefined if no arguments should be provided to the function.
```

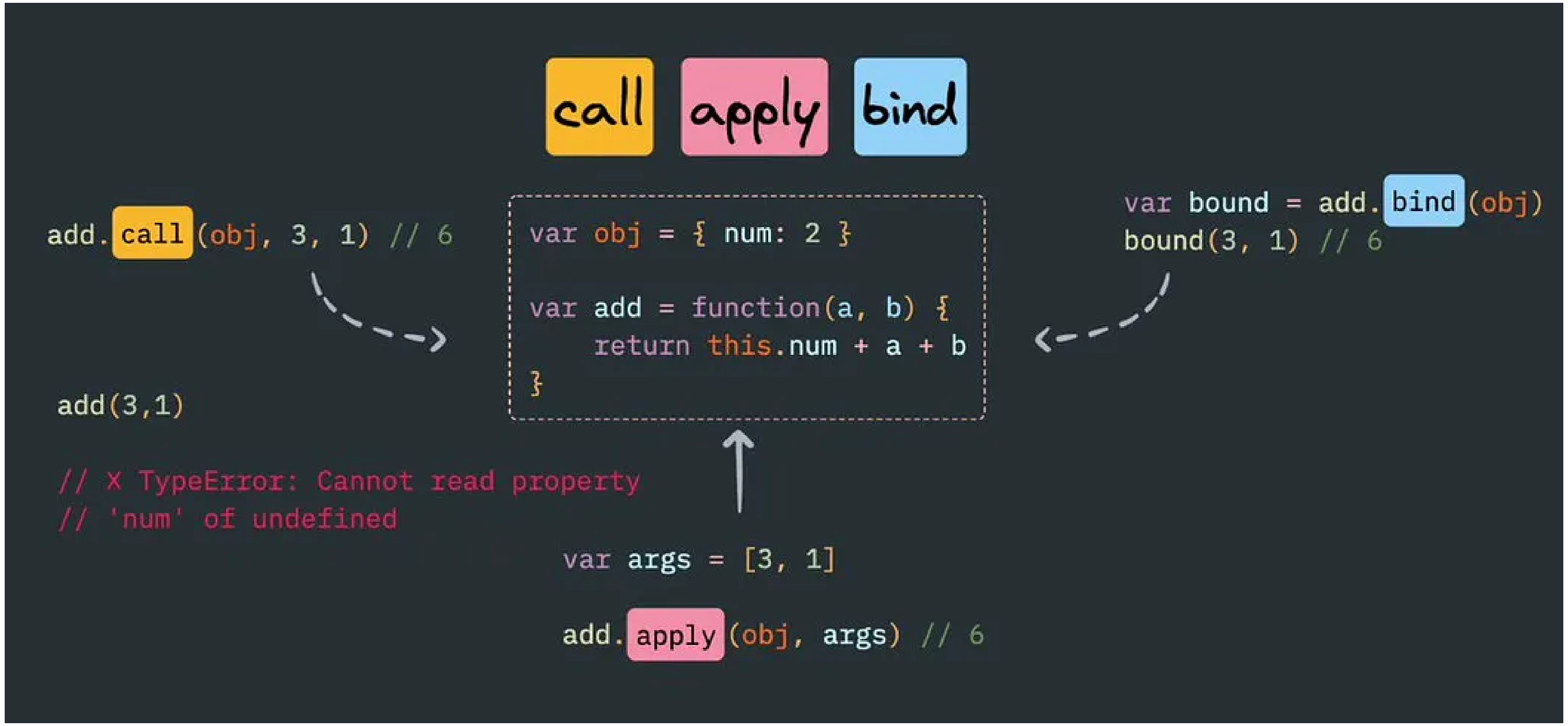
# Bind ( )

The **bind()** method returns a new function, when invoked, has its this sets to a specific value.



```
func.apply(thisArg, [argsArray]);

// Parameters:
// thisArg The value of this provided for the call to func.

// argsArray Optional. An array-like object, specifying the arguments with which func should be called,
// or null or undefined if no arguments should be provided to the function.
```

# Call/Bind/Apply - Summary



```
add.call(obj, 3, 1) // 6
```

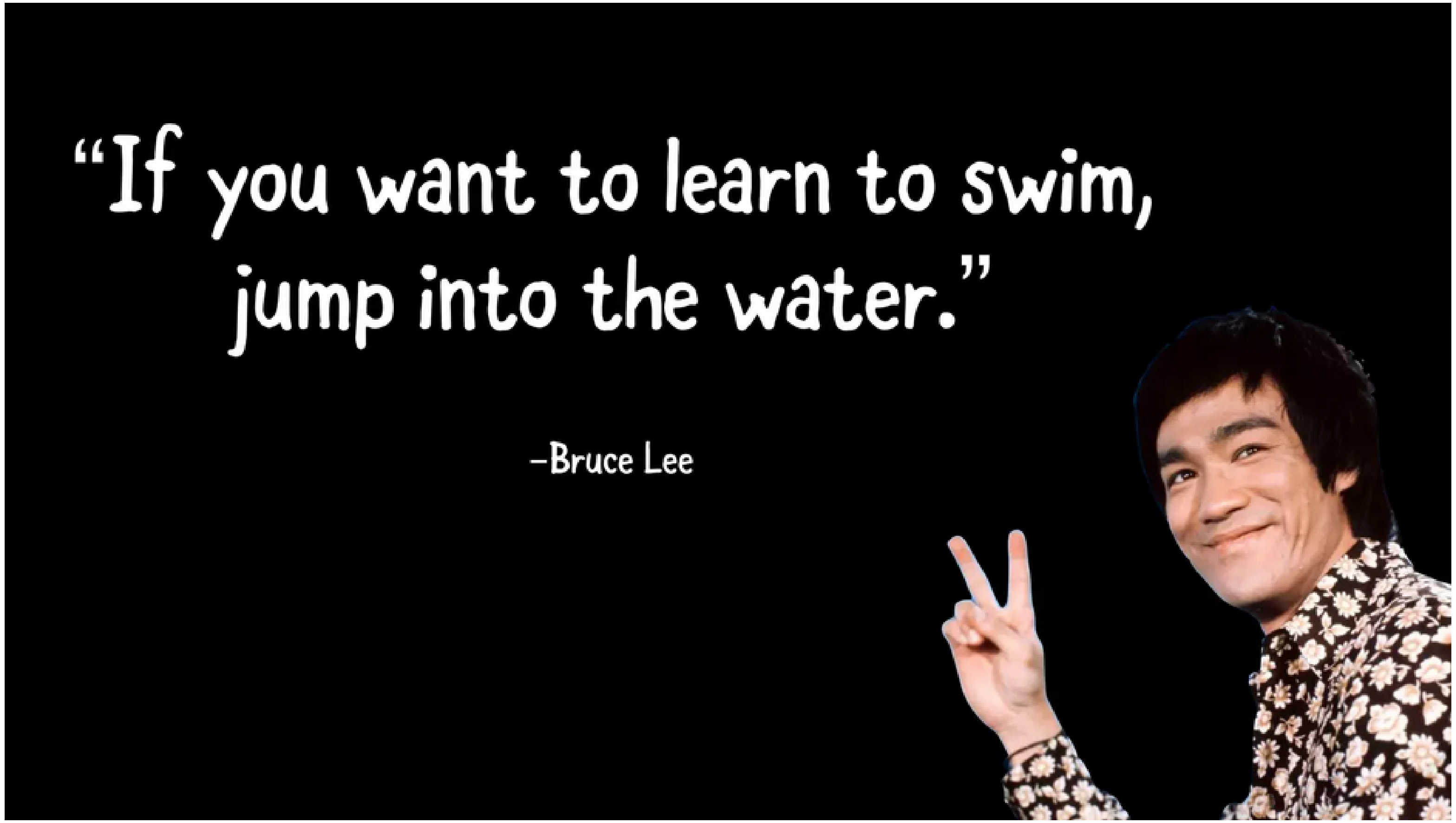
```
var obj = { num: 2 }

var add = function(a, b) {
    return this.num + a + b
}
```

```
var bound = add.bind(obj)
bound(3, 1) // 6
```

```
add(3,1)

// X TypeError: Cannot read property
// 'num' of undefined
```

```
var args = [3, 1]

add.apply(obj, args) // 6
```

# Call/Bind/Apply - Summary



“If you want to learn to swim, jump into the water.”

–Bruce Lee

# Q & A