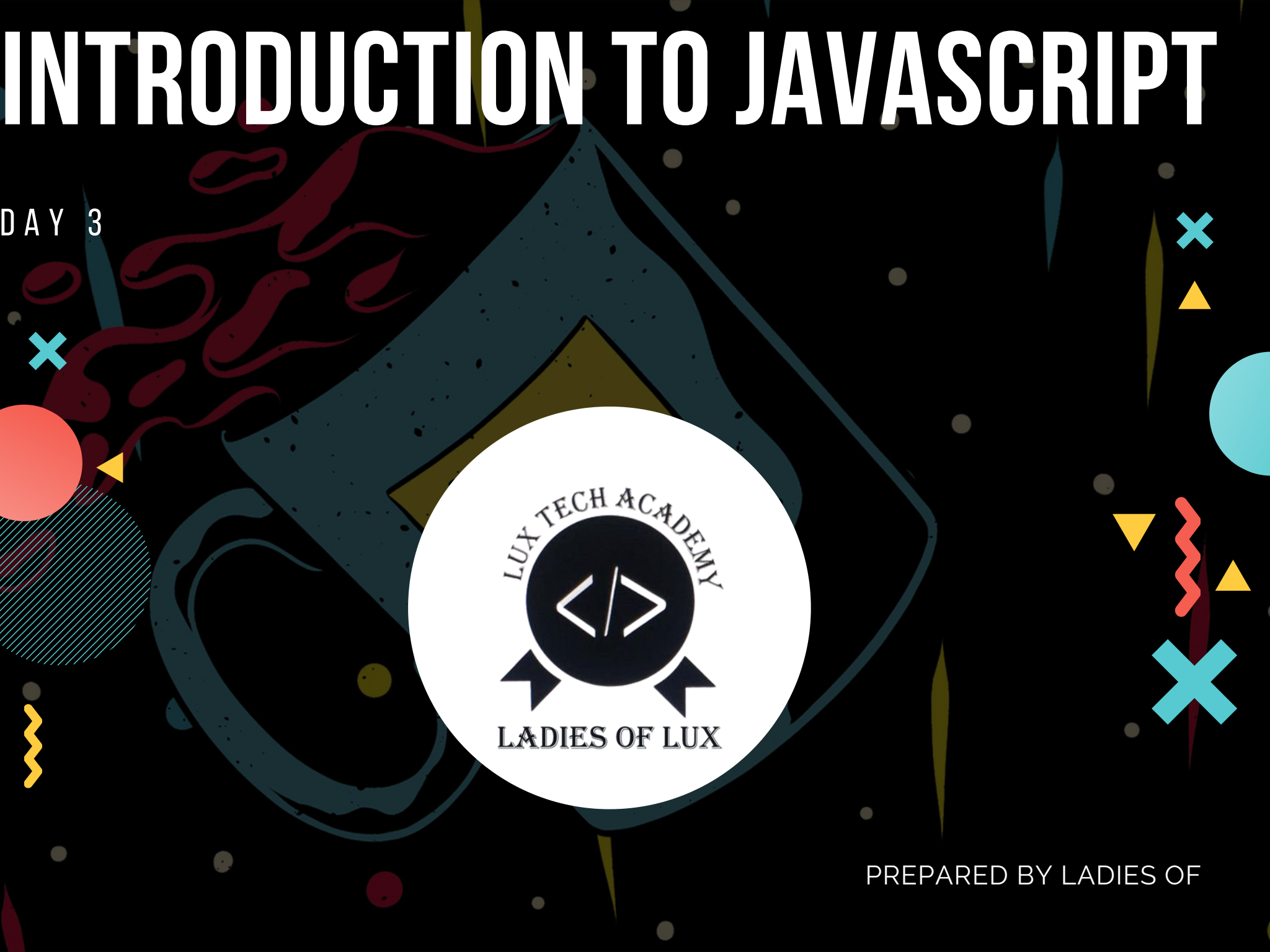
INTRODUCTION TO JAVASCRIPT

DAY 3



PREPARED BY LADIES OF

CONTENT

- Functions
- Scope
- Arrays
- Objects

JavaScript

FUNCTIONS

JavaScript Function

1. Function Definition
2. Function Parameters
3. Function Expressions
4. Arrow Functions
5. Regular Function
6. Generator Function
7. Function Declaration Vs Expression
8. Function Declaration in Conditional Statement

FUNCTIONS

- A JavaScript function is a block of code designed to
  perform a particular task. It is executed when something invokes it.

- A JavaScript function is defined with the function
  keyword followed by a name, followed by parentheses ().

FUNCTIONS

- The code to be executed by the function is placed inside curly braces {}
  - **Function parameters:**
    Listed inside the parenthesis in the function definition.
  - **Function arguments:**
    The values received by the function when it is invoked.

Function syntax:

```
function name (parameter1, parameter2, parameter3) {
  // code
to be executed
}
```

- **Returning a value**

  The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code.

Example 1

Write a function that returns the sum of two values:

```
function getSum(a,b) {
    //function returns the sum of a + b

    return a + b;
}
const result = getSum(4,5);
console.log(result);
```

Example 2

There may be occurrences of return in a single function

```
function checkAge(age){
    if(age <= 18){
        return confirm('Do you have permission from your parents
    }else{
        return true
    }
}
const age = prompt('How old are you ?')
if(checkAge(age)){
    alert('Access granted')
}else{
    alert('Access denied')
}
```

- **FUNCTION EXPRESSIONS**
  - **A** function expression can be stored in a variable. After a function expression has been stored in a variable, the variable can be used as a function.
  - A function expression can be used as **IIFE (Immediately Invoked Function Expression)** which runs as soon as it is defined.
  - The main difference between a function expression and a function declaration is the function name which can be omitted in function expressions to create anonymous functions.

Example:

```
const value = function(x,y){
    return x * y;
}
console.log(value(10, 20) );
```

FUNCTIONS AS VALUES

- Functions are not only syntax but also values, which means they can be assigned to variables, stored in properties of the object or elements of arrays, passed to arguments as arguments

Example: The function getSum is stored in the sum variable

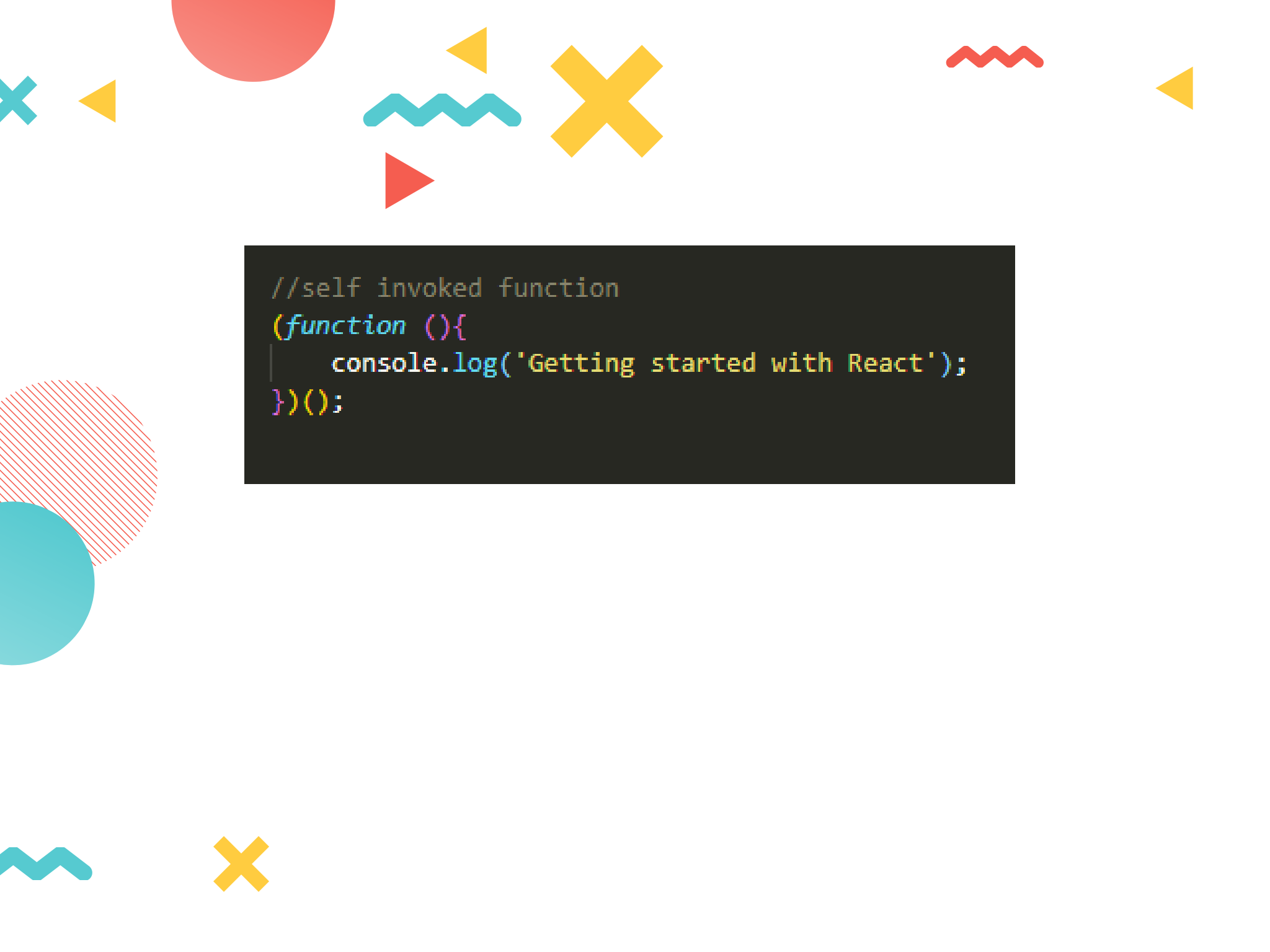
```
//function can be used as values
function getSum(a, b){
    return a + b;
}
const sum = getSum(5,6)
console.log(sum);
```

- **SELF_INVOKING FUNCTIONS**
  Function expressions can be made “self-invoking”

  A self-invoking expression is invoked (started) automatically, without being called.

  Function expressions will execute automatically if the expression is followed by ().

  You cannot self-invoke a function declaration.
  You have to add parentheses around the function to indicate that it is a function expression:


//self invoked function
(function (){
console.log('Getting started with React');
})();

JS Scope

SCOPE

Scope

SCOPE

Scope of a variable refers to the part of the code where it can be referenced and is "visible."

That is to say, a variable can only be referenced only within its scope.

JavaScript has a **global scope, local scope and lexical scope.**

- **Global scope**

  Variables with the global scope are visible in the entire application unless shadowed. Such variables are declared outside the functions or in functions but without the var keyword.

  Let's have an example

```
a="hello"
function greet(){
console.log(a)
}
greet()// outputs hello
```

Variable 'a' is declared outside any subroutine, hence has a global scope. Meaning, it is alive throughout the program.

Example 2;

In JavaScript, variables used without prior declaration also assume the global scope.
As it is the case of variable' name', in the below code.
Hence, it can be accessed at any part of the application.

```
function personsName(){
    name="John"
}
personsName()
console.log(name)
```

Note; note in the strict mode, a function cannot be accessed without declaring it first. Also, it is a good programming practice to minimize the use of global variables

- **Local scope**
  A variable can also have a local scope, that is, it can only be accessed within the function if it has been declared in the function block.

```
var c = "hello";

function greet1() {
    var b = "World"
}

 greet1();
console.log(c + b); // referenceError
```

The above program returns a referencing error because the variable b can only be accessed within the function scope.

- **Block-based scope- Let keyword**
  With ES6, the let keyword can be used to limit a variable only within its immediate block.
  Let's have an example

```
let b= "Hello"
function greetings(){
    if (b==="Hello"){
       let c="World"
    console.log(b + " "+c)
}
    console.log(c)
}
greetings()
```

Output: Hello World and a referenceError

Variable b, has a global scope, because it is defined outside any subroutine.

Variable c, has a local but block-based scope, that is to say, it can be accessed only within the if block but not even within the function.

Hence the statement console.log(c) returns a reference error.

- **Lexical scope**

  Lexical scope is found in the context of closures in JavaScript. Lexical scope simply means that the inner functions can access a variable defined in the outmost function, but vice versa is not true. Hence, variables declared in the inner function cannot be accessed in the outer functions.

  Below is an example for better understanding;

```
var a="Hello "
        function outerFunc(){
            var b="World. ";
               function innerFunc(){
                var c="Good morning";
                console.log( a+ b+ c)
            }
            innerFunc();
        }

        outerFunc()
```

The above code will output "Hello World. Good morning"

Var a is a global variable, hence it can be accessed anywhere in the code.

Var b, has a local scope, hence can be accessed only within the outerFunc().

However, the **outerFunc()** also declares the innerFunc(). Meaning, the inner function which is the innerFunc, can access the variables declared in the outer function, which is the outerFunc() function.

A step further, the **outerFunc**, can access the variable a, and because of the fact that the innerFunc has the privilege of accessing variables in outerFunc scope, also, it is aware of the var a.

Let's adjust the above code slightly;

```
var a="Hello "
    function outerFunc(){
     var b="World ";
     console.log(a + b+ c);
     function innerFunc(){
         var c="Good morning";

     }
     innerFunc();
 }

 outerFunc()
```

Can you guess the output?

It will be a referenceError. Why?

As per the above code variable c does not exist. Variable c is declared in the inner function, and referenced in the outer function.

Per the lexical scooping, the inner function can access the variables of the outer function, but the outer function cannot access variables of the inner function.

ARRAYS

Array

In JavaScript

ARRAYS

An array is a special variable which can store multiple values.

An array can hold many values under a particular name, the values can be accessed by reffering to an index number

[Array]

In Javascript

ARRAYS

Array Literal Syntax:

There are two syntax-es for creating an empty array

```
let array_name = new Array ();
let array_name = [];
```

Array

In Javascript

ARRAYS

Array elements are numbered according to zero.

```
const fruits = ["Apple", "Orange", "Plum"];

console.log(fruits[0] ); // Apple
console.log(fruits[1] ); // Orange
console.log(fruits[2] ); // Plum
```

ARRAYS

We can replace an element.
The statement changes the values of the third element in fruits

```
fruits[2] = 'Pear'; // now ["Apple", "Orange", "Pear"]
```

Adding a new item to the array

```
// Add a new one to the array:
fruits[3] = 'Lemon'; // now ["Apple", "Orange",
"Pear", "Lemon"]
```

ARRAYS

The total count of the elements in the array is its length:

```
// The length of the array
const fruits = ["Apple", "Orange",
"Plum"];

console.log( fruits.length ); // 3
```

[Array]

In Javascript

ARRAYS

The full array can be accessed by referring to the array name:

```
// Access the Full Array
const fruits = ["Apple", "Orange",
"Plum"];

console.log( fruits );
```

Accessing the last Array element

```
//Access the last element
const fruits = ["Banana", "Orange", "Apple", "Plum"];

const last = fruits[fruits.length - 1];

console.log(last);
```

ARRAY METHODS

- **Popping**

  Pop method extracts the last element of an array

```
const fruits = ["Apple", "Orange", "Pear"];

console.log( fruits.pop() ); // remove "Pear"
```

- **Pushing**

  Push () method append the element to the end of the array

  The push () method returns the new array length

```
const fruits = ["Apple", "Orange"];

fruits.push("Pear");

console.log( fruits ); // Apple, Orange, Pear
```

- **Shifting**

  The shift () method removes the first array element and shifts all other elements to a lower index.

```
const  fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.shift();  //Removes Banana from fruits

console.log(fruits);
```

- **Unshift**

  The Unshift () method adds a new element to an array at the beginning and unshifts older elements.

  The unshift method returns the new array length.

```
const fruits = ["Banana", "Orange", "Apple", "Mango"];

fruits.unshift("pawpaw");

console.log(fruits); // (5) ['pawpaw', 'Banana', 'Orange', 'Apple', 'Mango']
```

Other methods

toString () – Converts an array to a string of (comma separated) array values

splice () – used to add new items to an array. It returns an array with deleted items

concat () – creates a new array by merging existing arrays.

slice () – slices out a piece of an array into a new array.

sort () – Sorts an array alphabetically.

reverse () – reverses the element in an array. You can use it to sort an array in descending order.

ASSIGNMENT

- Do more coding challenges on https://www.codewars.com/?language=javascript

ARRAY ITERATION METHODS

- **Array.forEach()**

  The forEach () method calls a function (a callback function) once for each array element.

  Callback is invoked with three arguments
  - The value of the element
  - The index of the element
  - The Array object being traversed

- **Array.map()**

  The map () method creates a new array by performing a function n each array element.

  It does not change the original array.

```
const squares = [4,5,6,7];
console.log(squares);
const numSquare = squares.map(function(square){
    return square * 2;
})
console.log(numSquare);
```

- **Array.filter()**

  The filter () method creates a new array with array elements that passes a test implemented by the callback function.

  The filter () method iterates over each element of the array and pass each element to the callback function.

```
const words = ['react', 'Angular', 'Vue','JQuery','Django'];
const demo = words.filter(word => word.length > 6);
console.log(demo);
```

- **Array.reduce()**
  The reduce () method executes a reducer function that you provide on each element of the array, resulting in single output value.

  The reducer function takes four arguments.
  - **Accumulator**
  - **Current Value**
  - **Current Index**
  - **Source Array**

  The reducers function's returned value is assigned to the accumulator whose value is remembered across each iteration throughout the array and it becomes the resulting value

```
//This is the initial value
const initialValue = 0;

//Numbers array
const sales = [5, 10, 20,20,4];

//reducer method that takes in accumulator and next time

const reducer = function(accumulator, item){
    return accumulator + item

}
//we give the reducer method our reducer function and our
initial value

const total = sales.reduce(reducer, initialValue);
console.log(total);
```

OBJECTS

OBJECTS

A JavaScript object is an entity having state and behavior (properties and method.

Creating Objects in JavaScript

- **By object literal**
- **By creating instance of Object directly using new keyword**
- **By using an object constructor using new keyword**

OBJECTS

The name: values pairs in JavaScript object are called properties

The syntax for creating Object Literal:

```
const object = {
property1 : value1,
property2 : value2,
propertyN : valueN
}
```

OBJECTS

Accessing Object Properties

You can access object properties in two ways.

```
objectName.propertyName
or
objectName["propertyName"]
```

```
const student = {
    firstName: 'John',
    lastName: 'Doe',
    age: 27
}
console.log(student.firstName); //john
```

The JavaScript Keyword new

The following example creates a JavaScript object with the new keyword

```
const person = new Object();
person.firstName= 'James';
person.lastName ='Quick';
person.age = 34;
person.isAdmin = true

console.log(`${person.firstName} ${person.lastName}`); // James
Quick
```

Adding New Properties

You can add new properties to an existing object by simply giving it a value

```
const student = {
    firstName: 'John',
    lastName: 'Doe',
    age: 27
}
student.type='postgraduate';
console.log(`${student.firstName} is a ${student.type}`);
```

Deleting Properties

The delete keyword deletes a property from an object

```
const student = {
    firstName: 'John',
    lastName: 'Doe',
    age: 27
}

delete student.age;
console.log(student.age); //undefined
```

The output will be undefined since we have deleted the property age in the student object.

The this Keyword

The JavaScript this keyword refers to the object it belongs to.
It has different values depending on where it is used:

- **In a method, this refers to the owner of the object**
- **Alone, this refers to the global object**
- **In a function, in strict mode, this is undefined**
- **In a function, this refers to the global objects**
- **In a function definition, this refers to the owner of the function.**
- **In an event, this refers to the element that received the event**.

Object Methods

Objects can also have methods

Methods are stored in properties as function definitions

You can access object method with the following syntax:

```
objectName.methodName()
```

Example: In the example below this is the person1 object that owns the personage function:

```
const person1 ={
    firstName: 'John',
    lastName: 'Doe',
    age: 27,
    personAge: function(){
        if(this.age > 18) {
            return `${this.firstName} ${this.lastName} is
${this.age} years old`
        }
    }
}
console.log(person1.personAge() ); //john Doe is 27 years old
```

For ...in Loop

The for...in statement loops through the properties of an object

The block of code inside of the for...in loop will be executed once for each property

Example:

Create an object storing the salaries of an employee. Write the code to sum all salaries and store in the variable sum.

```
const salaries ={
    Doe: 100,
    Jeremy: 150,
    Pete : 200
}
let wage =0;
for( let key in salaries){
    wage= wage + salaries[key]
}
console.log(`The total is : ${wage}`); //The total is : 450
```

Object Constructor

An object constructor is a blueprint for creating many objects of the same type.

Objects of the same type are created by calling the constructor function with new keyword.

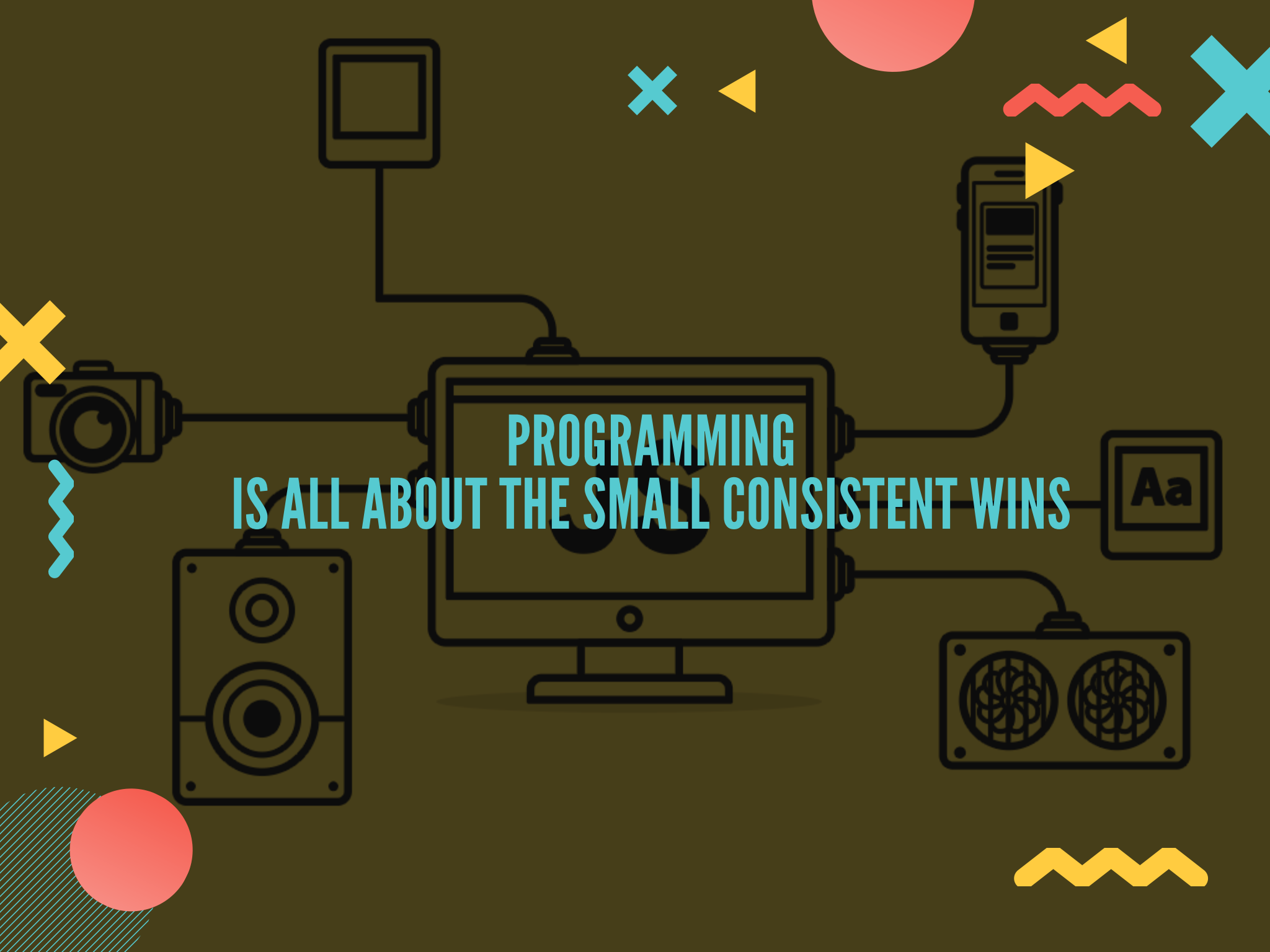
Adding methods to an object constructor must be done inside the constructor function.

```
function Employee(firstName, lastName,age, department){
    this.firstName = firstName;
    this.lastName = lastName;
    this.age= age;
    this.department = department
    this.changeDepartment = function(dept){
        this.department =dept;
    }

}
//Create an Employee Object
const employee1 = new Employee("John", "Rally", 32, "Design");

//Change the department
employee1.changeDepartment('Development');

 console.log(employee1.department); //Development
```


PROGRAMMING
IS ALL ABOUT THE SMALL CONSISTENT WINS
Aa

ALL THE BEST LADIES OF LUX

YOU GOT JOKES ,LOL
WHAT ARE YOU LOOKING FOR HERE , WE DONT HAVE BOTTOM JS