Anton Zamay
Posted on Mar 25, 2024 • Edited on Apr 2, 2024

❤️ 58 🦄 6 🤯 5 🙌 6 🔥 7

Lexical Scope, Lexical Environment, Execution Context, Closure in JavaScript

#javascript #scope #closure

In the world of JavaScript programming, mastering the fundamental concepts of Lexical Scope, Lexical Environment, Execution Context, Closures, and the `this` keyword is vital for crafting sophisticated and efficient code. These core principles govern how variables are accessed and functions are executed, enabling developers to write cleaner, more secure, and highly modular code. Understanding these concepts allows for better control over the scoping mechanism, execution flow, and the dynamic context of `this` within different scopes, laying the groundwork for advanced JavaScript programming practices and patterns.

TOC

- Lexical Scope
    - Global Scope
    - Local Scope
- Lexical Environment
    - Definition

    - Lexical Scope vs Lexical Environment
- Execution Context
    - Types of Execution Contexts
    - Components of an Execution Context
- Closure
- `this` Keyword
    - `this` in the Global Scope
    - `this` in the Function Calls
    - `this` with Arrow Functions
    - Manipulating this with bind(), call(), and apply()

Lexical Scope

Lexical Scope in programming, particularly in JavaScript, refers to the context in which variables and functions are accessible or visible. Lexical Scopes can be broadly classified into two categories: Global Scope and Local Scope.

Global Scope

When a variable is declared outside any function, it belongs to the Global Scope. This means it can be accessed and modified from any other part of the code, regardless of the context. Global variables are accessible throughout the lifespan of the application, making them readily available but also posing a risk of unintended modifications, which can lead to bugs.

Example:

```
var globalVar = "I am a global variable";

function accessGlobalVar() {
    // Accessing the global variable within a function
    console.log(globalVar);
}

accessGlobalVar(); // Output: I am a global variable
console.log(globalVar); // Output: I am a global variable
```

Local Scope

Local Scope, on the other hand, refers to variables declared within a function or a block (in case of `let` and `const` in ES6). These variables are only accessible within that function or block, making them hidden from the rest of the program. This encapsulation ensures that local variables are shielded from the unintended sideeffects of the outer scope. There are two main types of Local Scopes: Function Scope and Block Scope.

- **Function Scope**: Variables declared within a function using `var`, `let`, or `const` are scoped to the function.

Example:

```
function localFunctionScope() {
    var functionScopedVar = "I am a local variable";
    // Output: I am a local variable
    console.log(functionScopedVar);
}

localFunctionScope(); // Output: I am a local variable
// Uncaught ReferenceError: functionScopedVar is not defined
console.log(localVar);
```

- **Block Scope**: Introduced in ES6 with `let` and `const`, variables declared inside a block `{}` are only accessible within that block.

Example:

```
if (true) {
    let blockScopedVar = "I am block-scoped";
    const anotherBlockScopedVar = "So am I";
    var globalVar = "I am global!!!";
    console.log(blockScopedVar); // Accessible here
    console.log(anotherBlockScopedVar); // Accessible here
}

// Uncaught ReferenceError: blockScopedVar is not defined
console.log(blockScopedVar);
// Uncaught ReferenceError: anotherBlockScopedVar
// is not defined
console.log(anotherBlockScopedVar);
// Output: I am global!!!
console.log(globalVar);
```

Lexical Environment

Lexical Environment is a core part of the JavaScript engine's execution context and plays a crucial role in how variables and functions are scoped, accessed, and managed.

Definition

Lexical Environment, on the other hand, is a more concrete mechanism employed by JavaScript engines during the runtime to manage and access the variables based on the Lexical Scope. It is a part of the JavaScript execution context that consists of two significant parts:

1. **The Environment Record**: This is where the specific variables, constants, and functions defined within the Lexical Scope are actually stored.

2. **A reference to the outer environment**: This refers to the Lexical Environment of the parent scope. It allows for the "chain" through which JavaScript searches for variable values when a variable isn't found in the immediate Lexical Environment.

Lexical Environments are created every time a block of code or a function is executed, containing all the local variables and being linked to an outer Lexical Environment. This effectively forms a Scope Chain that determines how variable lookups occur during the execution of the code.

Lexical Scope vs Lexical Environment

Lexical Scope is a part of JavaScript's scoping mechanism — it's about the rules that govern how variable names are resolved in nested layers of code. Lexical Environment, in contrast, refers to the specific implementation where these rules are applied — the concrete structure used by the JavaScript engine during the execution of the code to store variables and function declarations and to keep track of their relationships according to the Lexical Scope.

Execution Context

In JavaScript, an Execution Context can be thought of as an abstract concept that holds information about the environment where the current code is being executed. It includes everything that is needed to track the execution of a block of code or a function, such as the value of `this`, variables, objects, and functions that are accessible at a given time.

Types of Execution Contexts

There are three main types of Execution Contexts in JavaScript:

1. **Global Execution Context (GEC):** This is the default or base execution context. It's where code that is not inside any function gets executed and where globally

scoped variables reside. There is only one Global Execution Context in a JavaScript program.

2. **Functional Execution Context (FEC):** This context is created by the execution of the code inside a function. Each function call creates a new FEC, and it includes the function's arguments, local variables, etc. It also has a reference to its outer environment, which helps in implementing lexical scope.

3. **Eval Execution Context:** Created by the execution of JavaScript code in an `eval` function, though its use is not recommended due to security and performance issues.

Components of an Execution Context

An Execution Context is composed of a few key components:

1. **Variable Environment:** This includes variables and functions declared within the current context. It essentially represents the Lexical Environment for functions and global variables.

2. **Lexical Environment:** This is where the magic of closures and scope chain becomes evident. The Lexical Environment is a structure that holds identifier-variable mapping. (Here, an "identifier" refers to the name of variables/functions, and the "variable" is the actual reference to objects/functions/data.) This environment also has a reference to the outer environment, which could either point to the Global environment or an outer function environment, enabling the scope chain.

3. **This Binding:** The value of `this` is determined and stored in the Execution Context. The value of `this` depends on how the function is called, and can refer to the global context, the current instance of an object (in the case of methods), or be set explicitly using functions like `call`, `apply`, or `bind`.

Example

To solidify the understanding, let's look at a concrete example:

```
let globalVar = "Welcome to the Global Context";

function outerFunction() {
    let outerVar = "I'm in the Outer Function";

    function innerFunction() {
        let innerVar = "I'm in the Inner Function";
        console.log(innerVar);
        console.log(outerVar);
        console.log(globalVar);
    }

    return innerFunction;
}

const inner = outerFunction(); // Execution Context for outerFunction is crea
inner(); // Execution Context for innerFunction is created and executed.
```

When `outerFunction` is called, a new Functional Execution Context is created for it. Inside `outerFunction`, another function `innerFunction` is declared, which, when called, has its own execution context. The `innerFunction` has access to variables declared in its own scope (`innerVar`), the `outerFunction`'s scope (`outerVar`), and the global scope (`globalVar`). This is made possible through the combination of Execution Contexts, Lexical Environments, and the Closure property that JavaScript functions exhibit.

The richness of JavaScript's execution model allows for complex, yet structured execution and scope management, with Closures and scopes being correctly managed through Execution Contexts and their associated Lexical Environments. Understanding how Execution Contexts work under the hood is fundamental to mastering JavaScript and unlocking its full potential.

Closure

A Closure occurs when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope. In simpler terms, a Closure gives you access to an outer function's scope from an inner function. This is possible in JavaScript due to how Lexical Scoping works, allowing functions to maintain references to their surrounding Lexical Environment.

To understand Closures, it's important to process two key points:

1. Every function in JavaScript has access to the scope in which it was created (Lexical Scoping).
2. JavaScript functions are first-class objects, meaning they can be returned from other functions and passed around like any other value.

When a function returns another function, the returned function will maintain a reference to its original Lexical Environment (where it was created). This is what forms a closure. Despite being executed outside their original scope, functions that are returned from other functions can access the variables of their parent function's scope.

Example

Suppose you want to create a counter object that allows you to increment, decrement, and retrieve the current count value, but you want to keep the count value private and not directly accessible from outside the object. This is where Closures come into play, allowing us to encapsulate the count variable within the scope of a function while exposing specific functionalities.

```
function createCounter() {
    let count = 0; // `count` is a private variable

    return {
        increment: function() {
            count++;
            console.log(count);
        },
        decrement: function() {
            count--;
            console.log(count);
        },
        getCount: function() {
            console.log(count);
            return count;
        }
    };
}

// Using the createCounter function to create a counter instance
const counter = createCounter();

counter.increment(); // Output: 1
counter.increment(); // Output: 2
counter.decrement(); // Output: 1
counter.getCount(); // Output: 1

// Trying to access the private `count` variable directly
console.log(counter.count); // Output: undefined
```

1. When the `createCounter` function is invoked, a new Execution Context is created. Within this Execution Context, a Lexical Environment is established where the `count` variable resides. This Lexical Environment includes not just `count`, but also

the inner functions `increment`, `decrement`, and `getCount`. The `count` variable is in the Lexical Scope of `createCounter`. This means it's not accessible from outside `createCounter`'s Lexical Environment directly, thus maintaining its privacy.
2. The inner functions (`increment`, `decrement`, `getCount`) each form a closure. A closure is formed when a function (defined within another function) keeps a reference to its parent's Lexical Environment, even after the parent function has completed execution. This is possible because, in JavaScript, functions carry a "backpack" (the closure) that contains references to variables present in their Lexical Environment at the time they were created. So, even though the Execution Context of `createCounter` is removed from the execution stack after it finishes executing, the `count` variable remains accessible to `increment`, `decrement`, and `getCount` because of their Closures over the Lexical Environment of `createCounter`.
3. The returned object from `createCounter` exposes only the `increment`, `decrement`, and `getCount` methods, without directly exposing the `count` variable itself. This method of encapsulation ensures that `count` can only be modified or accessed through the interfaces provided, adhering to principles of data privacy and encapsulation. This structure is determined by the Lexical Scope at the time of the function's definition, which dictates what variables are accessible within the function's body.

`this` Keyword

The `this` keyword in JavaScript is a powerful concept that refers to the object it belongs to, providing a way to give methods access to their owning object. Its value is determined by how a function is called.

`this` in the Global Scope

In the Global Scope, `this` refers to the global object. In a browser environment, the global object is `window`, whereas in Node.js, it is `global`.

```
console.log(this === window); // Output: true (in a browser environment)
```

`this` in Function Calls

The value of `this` inside a function depends on how the function is called:

- **Regular function call**: In the Global Scope, `this` will point to the global object (e.g., `window` in browsers). However, in strict mode, `this` will be `undefined` since strict mode prevents automatic binding to the global object.

```
function show() {
  console.log(this);
}
show(); // Output: Window (in a browser environment)
```

- **Method call**: When a function is called as a method of an object, `this` points to the object the method is called on.

```
const person = {
  name: 'Anton',
  greet() {
    console.log(`Hi, ${this.name}`);
  }
};

// `this` refers to the `person` object
person.greet(); // Output: "Hi, Anton"
```

`this` with Arrow Functions

Arrow functions do not have their own `this` binding. Instead, they capture the `this` value of the enclosing Lexical Scope. This behavior makes them particularly useful for callbacks.

```
const person = {
  name: 'Antonio',
  timer() {
    setTimeout(() => {
      console.log(this.name);
    }, 1000);
  }
};

person.timer(); // Output: "Antonio" (after 1 second)
```

In the above example, the arrow function inside `setTimeout` captures the `this` value from its surrounding Lexical Environment (in this case, the `person` object), illustrating how Lexical Scope influences the value of `this`.

Manipulating `this` with `bind()`, `call()`, and `apply()`

JavaScript provides methods to explicitly set the value of `this` for a function:

- **call()**: Calls the function with a given `this` value and arguments provided individually.

```
function introduce(language) {
  console.log(`I am ${this.name} and I love ${language}.`);
}

const user = { name: 'Anthony' };
// I am Anthony and I love JavaScript.
introduce.call(user, 'JavaScript');
```

- **apply()**: Similar to `call()`, but arguments are passed as an array.

```
// I am Anthony and I love JavaScript.
introduce.apply(user, ['JavaScript']);
```

- **bind()**: Creates a new function that, when called, has its `this` keyword set to the provided value, with a given sequence of arguments preceding any provided when the new function is called.

```
const boundIntroduce = introduce.bind(user);
// I am Anthony and I love JavaScript.
boundIntroduce('JavaScript').
```

These methods are particularly useful for controlling the Execution Context.