**EXECUION CONTEXT**

**What is the Execution Context?**

In JavaScript, the execution context is an abstract concept that holds information about the environment within which the current code is being executed. This environment includes things like variable and function declarations, the value of **this**, and the scope chain.

An execution context in JavaScript is an environment where the JavaScript code is evaluated and executed. It consists of:

* The scope chain
* The value of **this**
* Variable and function declarations

**Types of Execution Contexts**

1. Global Execution Context
2. Function Execution Context
3. Eval Execution Context
4. Script Execution Context
5. Block Execution Context (Local Execution Context)

**The Global Execution Context**

**What is it?**

The Global Execution Context is the default context where the entire JavaScript code starts its execution. It is associated with the global object, which is **window** in browsers and **global** in Node.js.

**Key Points**

* There is only one global execution context.
* Variables and functions declared in this context are globally accessible.
* It has two phases: the Creation Phase and the Execution Phase.

The Global Execution Context is the first execution context created when a JavaScript program runs. It is associated with the global object and has two phases: the Creation Phase and the Execution Phase.

**Creation Phase**

During the creation phase, the JavaScript engine sets up the environment for the code to be executed. This involves:

1. **Creation of the Global Object**: In a browser, this is the **window** object. In Node.js, it's the **global** object.
2. **Creation of the this Binding**: In the global context, **this** refers to the global object.
3. **Setting Up the Memory Space (Variable Object)**: This is where all the variables and function declarations are stored as properties.

**Execution Phase**

After the creation phase, the JavaScript engine starts executing the code line by line. During this phase:

1. **Assignment of Values**: Variables and function declarations are assigned their respective values.
2. **Execution of Code**: The code is executed line by line within the global context.

**Key Points**

* **One Global Context**: There is only one global execution context in a JavaScript program.
* **Global Scope**: Variables and functions declared in the global context are accessible throughout the entire program.
* **Variable Hoisting**: During the creation phase, variable and function declarations are "hoisted" to the top of their scope. This means you can use a variable or function before its declaration in the code.

**Example**

var globalVar = "I'm a global variable";

function globalFunction() {

console.log("Inside global function");

}

globalFunction(); // Call the function

console.log(globalVar); // Access the global variable

Let's look at an example to understand the Global Execution Context:

var name = "Alice"; // Global variable

function greet() {

console.log("Hello, " + name);

}

greet(); // Function call

In this example, the following steps occur:

1. **Creation Phase**:
   * The global object is created (**window** in browsers).
   * The **this** keyword is set to refer to the global object.
   * Memory space is set up for the variable **name** and the function **greet**.
2. **Execution Phase**:
   * The variable **name** is assigned the value **"Alice"**.
   * The **greet** function is defined.
   * The **greet** function is called, which logs **"Hello, Alice"** to the console.

**Visualization**

Here's a simple visualization of the global execution context:

1. **Before Execution**:
   * **Global Object**: { **name**: **undefined**, **greet**: **function** }
   * **this**: **window** (or **global** in Node.js)
2. **During Execution**:
   * **Global Object**: { **name**: **"Alice"**, **greet**: **function** }
   * **this**: **window** (or **global** in Node.js)

**Conclusion**

Understanding the Global Execution Context is crucial for grasping how JavaScript manages the scope and execution of code. It helps you see how variables and functions are stored and accessed throughout your program. As you move on to more complex concepts like function execution contexts and closures, this foundational knowledge will be very beneficial.

about memory phase, code execution phase and this in function section

**2. Function Execution Context**

**What is it?**

A Function Execution Context is created whenever a function is invoked. Each function in JavaScript gets its own execution context.

**Key Points**

* Created every time a function is called.
* Has its own scope chain, **this** value, and activation object (contains arguments and local variables).
* Variables and functions are hoisted (declared at the top of their scope).

**Example**

function sayHello(name) {

var greeting = "Hello, " + name;

console.log(greeting);

}

sayHello("Alice"); // Call the function with argument "Alice"

**3. Eval Execution Context**

**What is it?**

The Eval Execution Context is created when code is executed inside the **eval** function. This function allows the execution of a string of JavaScript code within the current scope.

**Key Points**

* Executes code within the scope from which it is called.
* Can access and modify variables in the current scope.

**Example**

var x = 10;

var y = 20;

function testEval() {

eval("var z = x + y; console.log(z);"); // Outputs: 30

}

testEval();

**4. Script Execution Context**

**What is it?**

The Script Execution Context is created for each **<script>** tag in an HTML document. Each script tag creates its own execution context, but they share the global execution context.

**Key Points**

* Each **<script>** tag has its own execution context.
* Scripts are executed sequentially in the order they appear in the HTML document.

**Example**

<!DOCTYPE html>

<html>

<head>

<title>Script Execution Context</title>

</head>

<body>

<script>

var scriptVar = "I'm from the first script";

console.log(scriptVar);

</script>

<script>

console.log(scriptVar); // Can access the variable from the first script

</script>

</body>

</html>

**5. Block Execution Context (Local Execution Context)**

**What is it?**

A Block Execution Context is created for block-scoped code (code inside **{}**). Introduced with ES6, it uses **let**, **const**, and **class** for block scoping.

**Key Points**

* Created for blocks within functions, loops, conditionals, etc.
* Variables declared with **let**, **const**, and **class** are block-scoped.
* Variables in the block scope are not accessible outside the block.

**Example**

function testBlockScope() {

if (true) {

let blockVar = "I'm a block-scoped variable";

console.log(blockVar); // Outputs: I'm a block-scoped variable

}

// console.log(blockVar); // Uncaught ReferenceError: blockVar is not defined

}

testBlockScope();

**Summary**

Understanding these execution contexts is crucial for knowing how JavaScript handles variable scoping, the **this** keyword, and the execution flow. Here’s a quick recap:

1. **Global Execution Context**: The default context, associated with the global object. Contains globally accessible variables and functions.
2. **Function Execution Context**: Created for each function call. Contains function-specific variables and arguments.
3. **Eval Execution Context**: Created when executing code inside **eval**. Shares scope with the calling context.
4. **Script Execution Context**: Each **<script>** tag creates its own context but shares the global context.
5. **Block Execution Context**: Created for block-scoped code using **let**, **const**, and **class**.

**Visualization of Execution Context**

Here is a simple visual representation of the execution context:

1. **Global Context**:
   * **var globalVar = "I'm a global variable";**
   * **function globalFunction() { ... }**
2. **Function Context** (for **sayHello** function):
   * **var name = "Alice";**
   * **var greeting = "Hello, Alice";**
3. **Block Context** (inside **if** block):
   * **let blockVar = "I'm a block-scoped variable";**

Remember, JavaScript manages these contexts to control the scope, execution, and lifetime of variables and functions in your code. Understanding them helps you write more predictable and bug-free code.

**Execution Context in Functions**

When a function is invoked in JavaScript, a new execution context is created specifically for that function. This execution context goes through two main phases: the **Memory Creation Phase** and the **Code Execution Phase**. Understanding these phases helps clarify how JavaScript handles function execution, variable scope, and the **this** keyword.

**Memory Creation Phase**

During the memory creation phase, the JavaScript engine sets up the environment for the function to be executed. This includes:

1. **Setting up the Activation Object (AO)**: This object holds the variables, function declarations, and arguments defined within the function.
2. **Hoisting**: Variable and function declarations are hoisted to the top of their scope. This means that the declarations are processed before any code is executed.

**Example**

function sayHello() {

var greeting = "Hello";

console.log(greeting);

}

sayHello();

In the memory creation phase for **sayHello**:

* An activation object is created.
* The variable **greeting** is added to the activation object with an initial value of **undefined** due to hoisting.

**Code Execution Phase**

During the code execution phase, the JavaScript engine executes the code line by line within the function. In this phase:

* Values are assigned to the variables declared within the function.
* The code inside the function is executed.

Continuing the example:

function sayHello() {

var greeting = "Hello"; // Assignment during code execution phase console.log(greeting); // Outputs: Hello

}

sayHello();

In the code execution phase for **sayHello**:

* The variable **greeting** is assigned the value **"Hello"**.
* The **console.log(greeting)** statement is executed, printing **"Hello"** to the console.

**The this Keyword in Functions**

The value of **this** within a function depends on how the function is called. Let's explore different scenarios:

1. **Global Context**: When a function is called in the global context, **this** refers to the global object (**window** in browsers, **global** in Node.js).

function globalFunction() {

console.log(this);

}

globalFunction(); // Outputs: window (or global in Node.js)

**Method Call**: When a function is called as a method of an object, **this** refers to the object that the method is called on.

const person = {

name: "Alice",

greet: function() {

console.log(this.name);

}

};

person.greet(); // Outputs: Alice

1. **Constructor Function**: When a function is used as a constructor with the **new** keyword, **this** refers to the newly created instance.

function Person(name) {

this.name = name;

}

const alice = new Person("Alice");

console.log(alice.name); // Outputs: Alice

1. **Arrow Functions**: Arrow functions do not have their own **this** context. Instead, they inherit **this** from the enclosing lexical context (the context in which the arrow function was defined).

const obj = {

name: "Alice",

regularFunction: function() {

console.log(this.name); // Outputs: Alice

},

arrowFunction: () => {

console.log(this.name); // Outputs: undefined (or the global object’s name in non-strict mode)

}

};

obj.regularFunction();

obj.arrowFunction();

**Summary**

1. **Memory Creation Phase**:
   * Sets up the activation object for the function.
   * Hoists variable and function declarations.
2. **Code Execution Phase**:
   * Assigns values to variables.
   * Executes the function's code line by line.
3. **this Keyword**:
   * Global context: **this** refers to the global object.
   * Method call: **this** refers to the object calling the method.
   * Constructor function: **this** refers to the new instance created.
   * Arrow function: **this** is inherited from the enclosing lexical context.

Understanding these concepts is crucial for mastering how functions work in JavaScript and how the **this** keyword behaves in different contexts.

In JavaScript, **var**, **let**, and **const** are used to declare variables, but they have different behaviors in terms of scope, hoisting, and mutability. Understanding these differences is crucial for writing clear and bug-free code.

**Scope**

**var**

* **Function Scope**: Variables declared with **var** are function-scoped. This means they are accessible within the function they are declared in, including nested functions, but not outside of that function.
* **Global Scope**: If declared outside of any function, **var** becomes a property of the global object (**window** in browsers).
* **var:** Has either function scope or global scope. A variable declared with var is accessible throughout the entire function it's declared in, or globally if outside any function. This can lead to issues if you're not careful, as variables declared with var can bleed into other parts of your code unintentionally.

function example() {

if (true) {

var x = 10;

}

console.log(x); // Outputs: 10

}

example();

console.log(x); // ReferenceError: x is not defined

**let and const**

* **Block Scope**: Variables declared with **let** and **const** are block-scoped. This means they are only accessible within the block they are declared in (enclosed by **{}**).
* **let and const:** Have block scope. Introduced in ES6, these keywords restrict variable accessibility to the block (like an if statement or a loop) where they are declared. This helps prevent accidental modification and improves code organization.

function example() {

if (true) {

let y = 20;

const z = 30;

}

console.log(y); // ReferenceError: y is not defined

console.log(z); // ReferenceError: z is not defined

}

example();

**Reassignment:**

* **var:** Allows you to reassign values to the variable throughout the code.
* **let:** Also allows reassignment, but only within the block where it's declared.
* **const:** Does not allow reassignment of the value after the initial assignment. This ensures that the variable always holds the same value, making your code more predictable.

**Hoisting**

**var**

* Variables declared with **var** are hoisted to the top of their scope. This means their declaration is processed before any code is executed, but their initialization remains in place. They are initialized with **undefined**.

console.log(a); // Outputs: undefined

var a = 5;

**let and const**

* Variables declared with **let** and **const** are also hoisted to the top of their scope, but unlike **var**, they are not initialized. Accessing them before their declaration results in a **ReferenceError**. This behavior is due to the "temporal dead zone" (TDZ).

console.log(b); // ReferenceError: Cannot access 'b' before initialization

let b = 10;

console.log(c); // ReferenceError: Cannot access 'c' before initialization

const c = 15;

**Declaration Phase:**

* **Hoisting:** This concept applies to var declarations. Variables declared with var are hoisted to the top of their scope (function or global). You can access them even before their declaration in the code, but they will have a value of undefined until actually assigned.
* **No Hoisting:** let and const are not hoisted. You cannot access them before their declaration in the code. Trying to do so will result in a ReferenceError.

Here's an analogy to understand the difference:

* Imagine a library. var declared variables are like books placed anywhere in the library. You can find them regardless of where you start searching, but they might be empty (undefined) until properly shelved (assigned a value).
* let and const declared variables are like books placed on specific shelves within a section (block). You can only find them by going to that specific section (block) after they are shelved (declared).

**Example:**

// Function Scope with var

function withVar() {

if (true) {

var x = 10;

}

console.log(x); // This will still log 10 even though x is declared inside the if block

}

withVar(); // Output: 10

// Block Scope with let

function withLet() {

if (true) {

let y = 20;

}

console.log(y); // This will result in a ReferenceError because y is not accessible outside the block

}

withLet(); // throws ReferenceError

**Mutability**

**var and let**

* Both **var** and **let** allow you to reassign values to the variable after its initial declaration.

var d = 20;

d = 25; // Reassignment is allowed

let e = 30;

e = 35; // Reassignment is allowed

**const**

* Variables declared with **const** cannot be reassigned after their initial declaration. This makes **const** useful for declaring constants or variables that should not change.

const f = 40;

f = 45; // TypeError: Assignment to constant variable

* However, if a **const** variable holds an object or an array, the contents of the object or array can be modified, but the variable itself cannot be reassigned.

const obj = { prop: 50 };

obj.prop = 55; // Allowed

console.log(obj.prop); // Outputs: 55

const arr = [60, 70];

arr.push(80); // Allowed

console.log(arr); // Outputs: [60, 70, 80]

obj = {}; // TypeError: Assignment to constant variable

arr = []; // TypeError: Assignment to constant variable

**Summary**

* **Scope**:
  + **var**: Function-scoped or globally scoped.
  + **let** and **const**: Block-scoped.
* **Hoisting**:
  + **var**: Hoisted and initialized with **undefined**.
  + **let** and **const**: Hoisted but not initialized, causing a temporal dead zone until the declaration is encountered.
* **Mutability**:
  + **var** and **let**: Variables can be reassigned.
  + **const**: Variables cannot be reassigned, but the contents of objects and arrays declared with **const** can be modified.

Understanding these differences helps in choosing the appropriate variable declaration keyword for different situations, leading to more predictable and maintainable code.

Here’s a summary of their differences:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Keyword** | **Scope** | **Hoisting** | **Can Be Reassigned** | **Can Be Redeclared** |
| var | Function | Yes | Yes | Yes |
| let | Block | No | Yes | No |
| const | Block | No | No | No |

Certainly! Let's delve into local scope and how **var**, **let**, and **const** behave within it. Local scope refers to the scope that is created within functions or blocks.

**Local Scope**

**var**

* **Function Scope**: When **var** is used inside a function, it is scoped to that function. This means it is not accessible outside of that function.

function example() {

var a = 10;

console.log(a); // Outputs: 10

}

example();

console.log(a); // ReferenceError: a is not defined

**Global Scope**: When **var** is declared outside of any function, it becomes a global variable and is attached to the global object (**window** in browsers).

**let and const**

* **Block Scope**: Variables declared with **let** and **const** are scoped to the block they are declared in, such as within **{}** braces in functions, loops, or conditionals.

function example() {

if (true) {

let b = 20;

const c = 30;

console.log(b); // Outputs: 20

console.log(c); // Outputs: 30

}

console.log(b); // ReferenceError: b is not defined

console.log(c); // ReferenceError: c is not defined

}

example();

**Function Scope Example**

function functionScopeExample() {

var x = 5;

let y = 10;

const z = 15;

console.log(x); // Outputs: 5

console.log(y); // Outputs: 10

console.log(z); // Outputs: 15

}

functionScopeExample();

console.log(x); // ReferenceError: x is not defined

console.log(y); // ReferenceError: y is not defined

console.log(z); // ReferenceError: z is not defined

**Block Scope Example**

function blockScopeExample() {

if (true) {

var a = 1; // Function-scoped

let b = 2; // Block-scoped

const c = 3; // Block-scoped

}

console.log(a); // Outputs: 1

console.log(b); // ReferenceError: b is not defined

console.log(c); // ReferenceError: c is not defined

}

blockScopeExample();

**Loops and Block Scope**

Using **let** or **const** inside loops ensures that the variable is scoped to the loop block, which helps avoid common pitfalls with closures in loops.

for (let i = 0; i < 3; i++) {

console.log(i); // Outputs: 0, 1, 2

}

console.log(i); // ReferenceError: i is not defined

In contrast, using **var** in loops makes the variable accessible outside the loop block, as **var** is function-scoped.

for (var j = 0; j < 3; j++) {

console.log(j); // Outputs: 0, 1, 2

}

console.log(j); // Outputs: 3

**Summary**

* **Local Scope**:
  + Variables declared within a function are scoped to that function (function scope).
  + Variables declared within a block (using **let** or **const**) are scoped to that block (block scope).
* **var**:
  + Function-scoped: Accessible throughout the function.
  + Not block-scoped: Accessible outside the block they are declared in within the same function.
* **let and const**:
  + Block-scoped: Only accessible within the block they are declared in.
  + Prevents issues with closures in loops by providing block scope.

Understanding local scope is crucial for avoiding variable conflicts and ensuring proper variable management in your JavaScript code. It helps in writing more predictable and maintainable code.