Execution Context

Execution context is a fundamental concept in JavaScript that plays a crucial role in how code is executed. It can be thought of as an environment in which JavaScript code is executed, and it includes various components that affect how the code behaves and what variables and functions are accessible at any given moment.

Types of Execution Context

There are three types of execution contexts in JavaScript:

Global Execution Context

This is the outermost execution context and is created when your JavaScript code begins to run. It includes the global object (which is window in web browsers and global in Node.js), and any variables and functions declared in the global scope are accessible within this context. This context persists throughout the entire runtime of your script.

Function Execution Context

Whenever a function is invoked, a new function execution context is created. This context includes the function's arguments, local variables declared within the function, and references to variables from outer (enclosing) scopes (closures). Each time a function is called, a new function execution context is created, forming a stack of contexts (known as the "call stack") as function calls nest within each other.

Eval Execution Context

If the eval function is used to execute code dynamically, a special execution context called the "eval context" is created. This context is more restricted in some ways compared to normal function contexts due to the dynamic nature of the evaluated code.

Components of Execution Context

Each execution context includes the following components:

Variable Object (VO) / Lexical Environment (LE)

This component contains all the variables, functions, and function parameters available within the context. In modern JavaScript environments (ES6+), the term "Lexical Environment" is preferred. It represents the scope and the identifiers defined within that scope.

Scope Chain

The scope chain is a list of Lexical Environments that are searched when resolving variables. This allows nested functions to access variables from their containing (enclosing) functions.

This Value

The this value refers to the object that a function is a method of, or the global object (window in browsers, global in Node.js) if the function is not a method of any object. The value of this can change based on how a function is called.

Phases of Execution Context

An execution context in JavaScript goes through two main phases: the creation phase and the execution phase. These phases are essential for understanding how JavaScript code is processed and how variables and functions are organized and made available during execution.

Creation Phase

In this phase, the JavaScript engine prepares for the execution of code by setting up the environment for variables, functions, and this. The creation phase sets up the foundation for the upcoming execution.

Key tasks during the creation phase include:

Creating the Variable Object (VO) / Lexical Environment (LE)

This includes setting up the scope and environment for variables and functions. In modern JavaScript environments, the term "Lexical Environment" is preferred.

Hoisting

Function declarations and variable declarations are moved to the top of their respective scopes. This means they are available for use before they are explicitly defined in the code.

Creating the Scope Chain

This establishes the hierarchy of scopes in which variables can be accessed.

Determining the value of the this keyword

The value of this is established based on the function's context and how it's called.

Execution Phase

Once the creation phase is complete, the JavaScript engine moves to the execution phase, where the code is executed line by line.

Variables are assigned values, functions are called, and operations are carried out.

The order of execution is determined by the sequence of statements in the code.

The execution phase continues until the entire code has been executed or until a return statement or an exception occurs.

```javascript
console.log(x); // Output: undefined
var x = 5;

foo(); // Output: "Hello, world!"
function foo() {
    console.log("Hello, world!");
}
```

In the example above, during the creation phase:

The variable x is declared and hoisted with an initial value of undefined.
The function foo is declared and hoisted.

During the execution phase:

The console.log(x); statement outputs undefined because of hoisting.
The x = 5; assignment gives x a value of 5.
The foo(); statement calls the foo function, which logs "Hello, world!".

Call Stack

The call stack is a data structure that tracks the order in which function contexts are created and executed. It keeps track of the currently executing context and ensures that the correct context is completed before moving on to the next. The call stack follows the Last-In-First-Out (LIFO) principle, meaning the last function called is the first one to be completed.

As functions are called, their contexts are pushed onto the call stack. When a function completes its execution, its context is popped off the stack, and control returns to the previous context. This process continues until the stack is empty, which indicates that the script has finished executing.

```
function outer()
{
  console.log("Outer function");
  inner();
}

function inner()
{
  console.log("Inner function");
}

outer();
```

Call stack sequence (simplified):

1. Global Execution Context is created.
2. outer() is called, its context is pushed onto the stack.
3. outer() logs and calls inner().
4. inner() is called, its context is pushed onto the stack.
5. inner() logs.
6. inner() completes, its context is popped off the stack.
7. Control returns to the outer() context.
8. outer() completes, its context is popped off the stack.
9. Global Execution Context remains.