Global Execution Context:

The Global Execution Context is the default context in which the JavaScript code runs. It has two main phases: the Memory Allocation Phase (Creation Phase) and the Execution Phase. Let's use the given code example:

```
let globalVar = "This is a global variable";

function sayHello() {
  let functionVar = "This is a function variable";
  console.log(globalVar); // Accessing global variable

  function nestedFunction() {
    const nestedConst = "This is a nested constant";
    console.log(functionVar); // Accessing function variable
    console.log(globalVar);  // Accessing global variable
  }

  nestedFunction();
}

sayHello();
```

1. Global Execution Context: Memory Allocation Phase (Creation Phase)

During the Memory Allocation Phase, the JavaScript engine allocates memory for variables and functions. Variables declared with let and const are placed in the Temporal Dead Zone (TDZ) until they are initialized. Functions are hoisted with their entire definitions (i.e., their code bodies).

Memory Allocation Phase:

1. Global Execution Context Memory:
    - `globalVar`: Not initialized (in TDZ)
    - `sayHello`: Reference to the entire function definition

Memory State:

```
globalVar: <uninitialized> (TDZ)
sayHello: function sayHello() {
  let functionVar = "This is a function variable";
  console.log(globalVar);

  function nestedFunction() {
    const nestedConst = "This is a nested constant";
    console.log(functionVar);
    console.log(globalVar);
  }

  nestedFunction();
}
```

2. Global Execution Context: Execution Phase:

In the Execution Phase, the JavaScript engine executes the code line by line and initializes variables.

Execution Phase:

```
globalVar = "This is a global variable"; // Now, globalVar is assigned the string value.
```

Memory State after Assignment:

```
globalVar: "This is a global variable"
sayHello: function sayHello() { ... }
```

When `**sayHello()**` is called, a new Execution Context for the sayHello function is created.

Function Execution Context (sayHello)

Each function creates its own Execution Context, which also has Memory Allocation and Execution Phases.

1. Function Execution Context: Memory Allocation Phase (Creation Phase)

Memory Allocation Phase for sayHello:

1. **Function Execution Context Memory:**
   - **`functionVar`:** Not initialized (in TDZ)
   - **`nestedFunction`:** Reference to the entire function definition

Memory State:

```
functionVar: <uninitialized> (TDZ)
nestedFunction: function nestedFunction() {
  const nestedConst = "This is a nested constant";
  console.log(functionVar);
  console.log(globalVar);
}
```

2. Function Execution Context: Execution Phase:

Execution Phase for sayHello:

```
functionVar = "This is a function variable"; // Now, functionVar is assigned the string
console.log(globalVar); // Logs: "This is a global variable"
```

When `nestedFunction` is called, a new Execution Context for the `nestedFunction` is created.

Function Execution Context (nestedFunction)

1. Function Execution Context: Memory Allocation Phase (Creation Phase)

Memory Allocation Phase for nestedFunction:

1. **Function Execution Context Memory:**
    - `nestedConst`: Not initialized (in TDZ)

Memory State:

```
javascript

nestedConst: <uninitialized> (TDZ)
```

2. Function Execution Context: Execution Phase

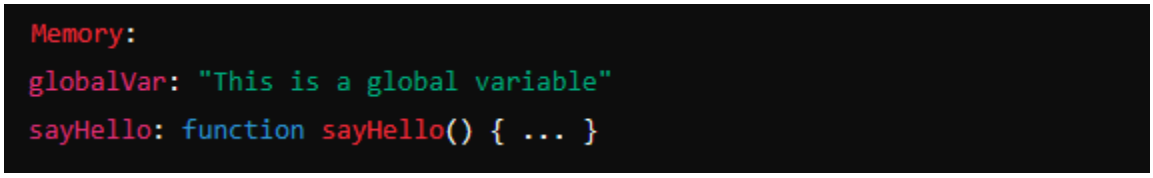
Execution Phase for nestedFunction:

```
nestedConst = "This is a nested constant"; // Now, nestedConst is assigned the string value.
console.log(functionVar); // Logs: "This is a function variable"
console.log(globalVar);  // Logs: "This is a global variable"
```

Detailed Step-by-Step Explanation:

1. **Global Execution Context: Memory Allocation Phase**
    - `nestedConst`: Not initialized (in TDZ)
    - `function sayHello()` is hoisted with its entire definition (including its code body)

2. **Global Execution Context: Execution Phase**
    - `globalVar` is assigned `"This is a global variable"`.



3. **sayHello Execution Context: Memory Allocation Phase**
    - `let functionVar;` is declared but not initialized (in TDZ).
    - `function nestedFunction()` is hoisted with its entire definition (including its code body).

```
Memory:
functionVar: <uninitialized> (TDZ)
nestedFunction:
  ┌──────────────────────────┐
  │ function nestedFunction() {│
  │   const nestedConst = "This│
  │   console.log(functionVar);│
  │   console.log(globalVar);  │
  │ }                          │
  └──────────────────────────┘
```

4. **sayHello Execution Context: Execution Phase**
    - `functionVar`; is assigned "This is a function variable".
    - Logs: `"This is a function variable"

```
Memory:
functionVar: "This is a function variable"
nestedFunction: function nestedFunction() { ... }
```

5. **nestedFunction Execution Context: Memory Allocation Phase**
    - `const nestedConst`; is declared but not initialized (in TDZ).

```
Memory:
nestedConst: <uninitialized> (TDZ)
```

6. **nestedFunction Execution Context: Execution Phase**
    - `nestedConst`; is assigned "This is a nested constant".
    - Logs: `"This is a function variable"
    - Logs: `"This is a global variable"

```
Memory:
nestedConst: "This is a nested constant"
```

This detailed breakdown accurately represents the memory states and execution contexts at each step of the code, making it clear how JavaScript handles variable and function

declarations and executions, particularly regarding the Temporal Dead Zone for `let` and `const` declarations, and the hoisting of functions with their entire definitions.