15.1. Introduction

In the Functions chapter, we saw that *where* variables are declared and initialized in the code affects when they can be used. This idea is called **scope**, and it describes the ability of a program to access or modify a variable.

Example

```js
let a = 0;

function coolFunction() {
   let b = 2;
   return a + b;
}
```

`a` is accessible *inside* and *outside* of `coolFunction()`.

`b` is only accessible *inside* of `coolFunction()`.

Let's add some `console.log` statements to explore this code snippet.

Example

```js
1  let a = 0;
2  console.log(a);
3
4  function coolFunction() {
5     let b = 2;
6     console.log(`a = ${a}, b = ${b}.`);
7     return a + b;
8  }
9
10 a += 1;
11 console.log(a);
12
13 coolFunction();
14 console.log(b);
```

Console Output

```
0
1
a = 1, b = 2.
ReferenceError: b is not defined
```

1. Lines 2 and 11 print the initial and incremented values of `a`.
2. Line 13 calls `coolFunction()`, and line 6 prints the values of `a` and `b`. This shows that both variables are accessible within the function.
3. Line 14 throws a ReferenceError, showing that `b` is not accessible outside of `coolFunction`.

15.1.1. Block/Local Scope

Local scope refers to variables declared and initialized inside a function or block. A *locally scoped* variable can only be referenced inside of the block or function where it is defined. In the example above, `b` has a local scope limited to `coolFunction()`. Referencing or attempting to update `b` outside of the function leads to a scoping error.

Try It!

The following code block has an error related to scope. Try to fix it!

```javascript
function myFunction() {
   let i = 10;
   return 10 + i;
}

console.log(i);
```

repl.it

15.1.2. Global Scope

Global scope refers to variables declared and initialized outside of a function and in the main body of the file. These variables are accessible to any function within a file. In the first example above, `a` has global scope.

Global scope is the default in JavaScript. If you assign a value to a variable WITHOUT first declaring it with `let` or `const`, then the variable automatically becomes global.

Example

```javascript
// Code here CAN use newVariable.

function coolFunction() {
   newVariable = 5;
   return newVariable;
}

// Code here CAN use newVariable.
```

Warning

In the loop `for (i = 0; i < string.length; i++)`, leaving off the `let` from `i = 0` means that `i` is treated as a global variable. ANY other portion of the program can access or modify `i`, which could disrupt how well the loop operates.

15.1.3. Execution Context

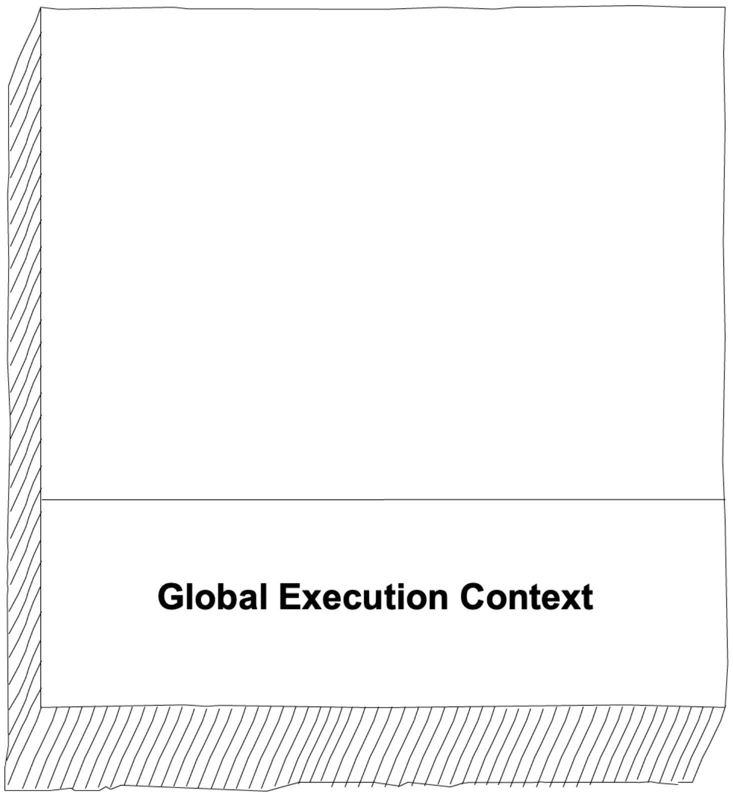
Execution context refers to the conditions under which a variable is executed—its scope. Scoping affects the variable's behavior at runtime. When the code is run in the browser, everything is first run at a global context. As the compiler processes the code and finds a function, it shifts into the function context before returning to global execution context.

Let's consider this code:

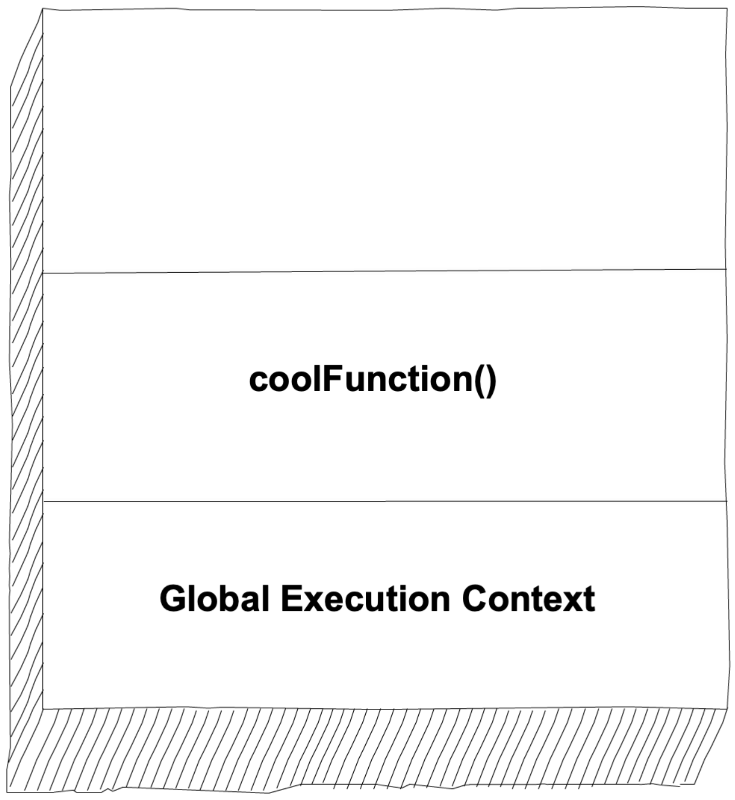
```javascript
let a = 0;

function coolFunction() {
   let b = 0;
   return a + b;
}

function coolerFunction() {
  let c = 0;
  c = coolFunction();
  return c;
}
```

Now, let’s consider the execution context for each step.

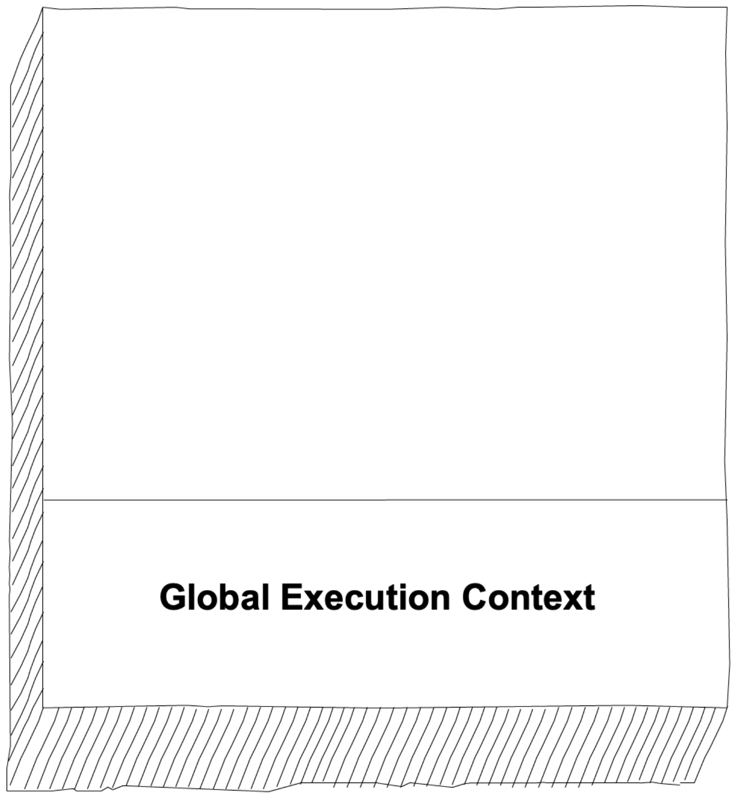
1. First, the global execution context is entered as the compiler executes the code.



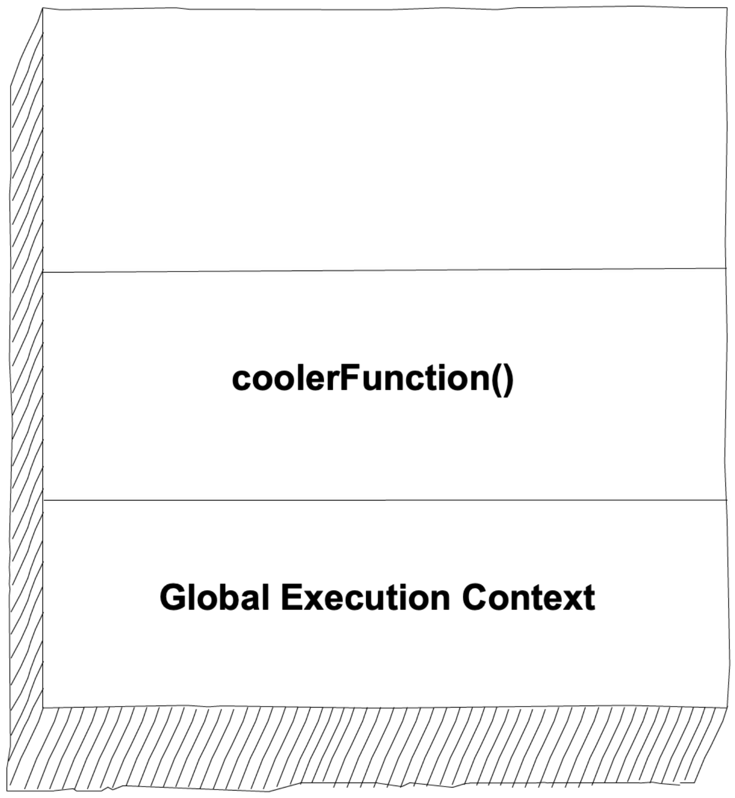
2. Once `coolFunction()` is hit, the compiler creates and executes `coolFunction()` under the `coolFunction()` execution context.



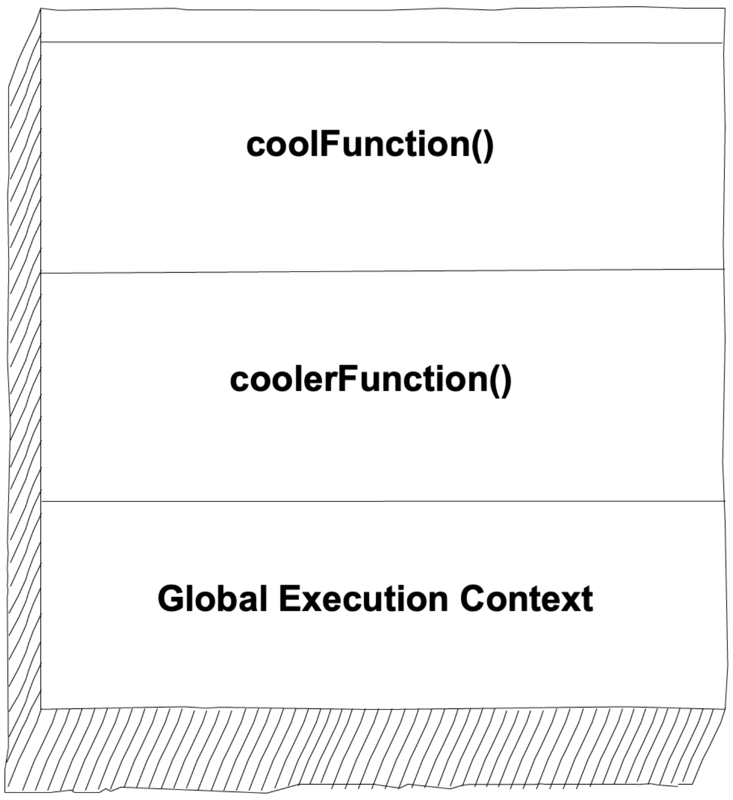
3. Upon completion, the compiler returns to the global execution context.

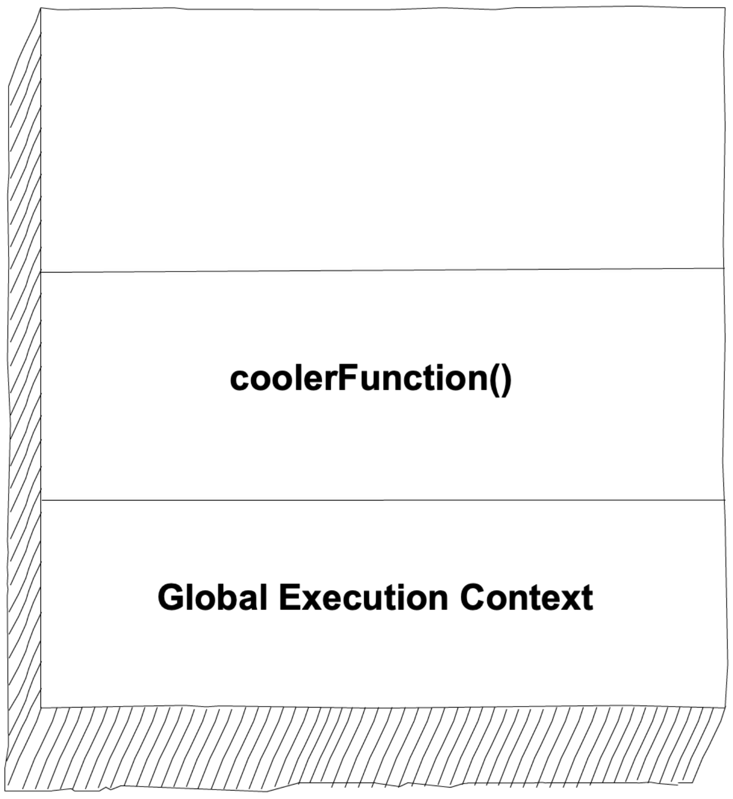


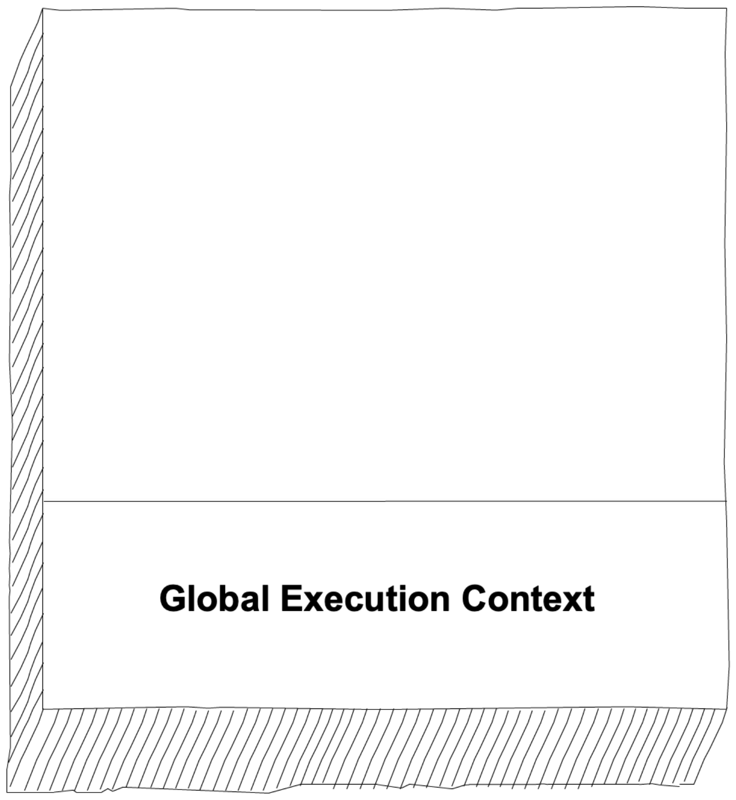
4. The compiler stays at the global execution context until the creation and execution of `coolerFunction()`.



5. Inside of `coolerFunction()` is a call to `coolFunction()`. The compiler will go up in execution context to `coolFunction()` before returning down to `coolerFunction()`’s execution context. Upon completion of that function, the compiler returns to the global execution context.


coolFunction()
coolerFunction()
Global Execution Context


coolerFunction()
Global Execution Context



15.1.4. Check Your Understanding

Both of the concept checks refer to the following code block:

```
function myFunction(n) {
   let a = 100;
   return a + n;
}

let x = 0;

x = myFunction(x);
```

Question

What scope is variable `x`?

a. Global
b. Local

Question

In what order will the compiler execute the code?

15.2. Using Scope

Scope allows programmers to control the flow of information through the variables in their program. Some variables you want to set as constants (like pi), which can be accessed globally. Others you want to keep secure to minimize the danger of accidental updates. For example, a variable holding someone's username should be kept secure.

15.2.1. Shadowing

Variable shadowing is where two variables in different scopes have the same name. The variables can then be accessed under different contexts. However, shadowing can affect the variable's accessibility. It also causes confusion for anyone reviewing the code.

Example

```
 1  const input = require('readline-sync');
 2  
 3  function hello(name) {
 4     console.log('Hello,', name);
 5     name = 'Ruth';
 6     return doubleName(name);
 7  }
 8  
 9  function doubleName(name){
10     console.log(name+name);
11     return name+name;
12  }
13  
14  let name = input.question("Please enter your name: ");
15  
16  hello(name);
17  doubleName(name);
18  console.log(name);
```

So, what is the value of `name` in line 4, 10, 16, 17, and 18?

Yikes! This is why shadowing is NOT a best practice in coding. Whenever possible, use different global and local variable names.

Try It!

If you are curious about the `name` values in the example, feel free to run the code here.

15.2.2. Variable Hoisting

Variable hoisting is a behavior in JavaScript where variable declarations are raised to the top of the current scope. This results in a program being able to use a variable before it has been declared. Hoisting occurs when the `var` keyword is used in the declaration, but it does NOT occur when `let` and `const` are used in the declaration.

Note

Although we don't use the `var` keyword in this book, you will see it a lot in other JavaScript resources. Variable hoisting is an important concept to keep in mind as you work with JavaScript.

15.2.3. Check Your Understanding

Question

What keyword allows a variable to be hoisted?

a. `let`
b. `var`
c. `const`

Question

Consider this code:

```
let a = 0;

function myFunction() {
   let a = 10;
   return a;
}
```

Because there are two separate variables with the name, `a`, under the two different scopes, `a` is being shadowed.

a. True
b. False