Differences between var,let,const keywords:

Var

Before the advent of ES6, `var` declarations ruled. There are issues associated with variables declared with `var`, though. That is why it was necessary for new ways to declare variables to emerge. First, let's get to understand `var` more before we discuss those issues.

Scope of var

Scope essentially means where these variables are available for use. `var` declarations are globally scoped or function/locally scoped.

The scope is global when a `var` variable is declared outside a function. This means that any variable that is declared with `var` outside a function block is available for use in the whole window.

`var` is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.

To understand further, look at the example below.

```
var greeter = "hey hi";

function newFunction() {
    var hello = "hello";
}
```

Here, `greeter` is globally scoped because it exists outside a function while `hello` is function scoped. So we cannot access the variable `hello` outside of a function. So if we do this:

```
var tester = "hey hi";
```

```
function newFunction() {
    var hello = "hello";
}
console.log(hello); // error: hello is not defined
```

We'll get an error which is as a result of `hello` not being available outside the function.

var variables can be re-declared and updated

This means that we can do this within the same scope and won't get an error.

```
var greeter = "hey hi";
var greeter = "say Hello instead";
```

and this also

```
var greeter = "hey hi";
greeter = "say Hello instead";
```

Hoisting of var

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that if we do this:

```
console.log (greeter);
var greeter = "say hello"
```

it is interpreted as this:

```
var greeter;
console.log(greeter); // greeter is undefined
greeter = "say hello"
```

So `var` variables are hoisted to the top of their scope and initialized with a value of `undefined`.

Problem with var

There's a weakness that comes with `var`. I'll use the example below to explain:

```
var greeter = "hey hi";
var times = 4;
```

```
if (times > 3) {
    var greeter = "say Hello instead";
}

console.log(greeter) // "say Hello instead"
```

So, since `times > 3` returns true, `greeter` is redefined to `"say Hello instead"`. While this is not a problem if you knowingly want `greeter` to be redefined, it becomes a problem when you do not realize that a variable `greeter` has already been defined before.

If you have used `greeter` in other parts of your code, you might be surprised at the output you might get. This will likely cause a lot of bugs in your code. This is why `let` and `const` are necessary.

Let

`let` is now preferred for variable declaration. It's no surprise as it comes as an improvement to `var` declarations. It also solves the problem with `var` that we just covered. Let's consider why this is so.

let is block scoped

A block is a chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block.

So a variable declared in a block with `let` is only available for use within that block. Let me explain this with an example:

```
let greeting = "say Hi";
let times = 4;

if (times > 3) {
    let hello = "say Hello instead";
    console.log(hello);// "say Hello instead"
 }
console.log(hello) // hello is not defined
```

We see that using `hello` outside its block (the curly braces where it was defined) returns an error. This is because `let` variables are block scoped .

let can be updated but not re-declared.

Just like `var`, a variable declared with `let` can be updated within its scope. Unlike `var`, a `let` variable cannot be re-declared within its scope. So while this will work:

```
let greeting = "say Hi";
greeting = "say Hello instead";
```

this will return an error:

```
let greeting = "say Hi";
let greeting = "say Hello instead"; // error: Identifier 'greeting' has already been declared
```

However, if the same variable is defined in different scopes, there will be no error:

```
let greeting = "say Hi";
if (true) {
    let greeting = "say Hello instead";
    console.log(greeting); // "say Hello instead"
}
console.log(greeting); // "say Hi"
```

Why is there no error? This is because both instances are treated as different variables since they have different scopes.

This fact makes `let` a better choice than `var`. When using `let`, you don't have to bother if you have used a name for a variable before as a variable exists only within its scope. Also, since a variable cannot be declared more than once within a scope, then the problem discussed earlier that occurs with `var` does not happen.

Hoisting of let

Just like `var`, `let` declarations are hoisted to the top. Unlike `var` which is initialized as `undefined`, the `let` keyword is

not initialized. So if you try to use a `let` variable before declaration, you'll get a `Reference Error`.

Const

Variables declared with the `const` maintain constant values. `const` declarations share some similarities with `let` declarations.

const declarations are block scoped

Like `let` declarations, `const` declarations can only be accessed within the block they were declared.

const cannot be updated or re-declared

This means that the value of a variable declared with `const` remains the same within its scope. It cannot be updated or re-declared. So if we declare a variable with `const`, we can neither do this:

```
const greeting = "say Hi";
greeting = "say Hello instead";// error: Assignment to constant variable.
```

nor this:

```
const greeting = "say Hi";
const greeting = "say Hello instead";// error: Identifier 'greeting' has already been declared
```

Every `const` declaration, therefore, must be initialized at the time of declaration.

This behavior is somehow different when it comes to objects declared with `const`. While a `const` object cannot be updated, the properties of this objects can be updated. Therefore, if we declare a `const` object as this:

```
const greeting = {
    message: "say Hi",
    times: 4
}
```

while we cannot do this:

```
greeting = {
    words: "Hello",
    number: "five"
```

```
} // error: Assignment to constant variable.
```

we can do this:

```
greeting.message = "say Hello instead";
```

This will update the value of `greeting.message` without returning errors.

Hoisting of const

Just like `let`, `const` declarations are hoisted to the top but are not initialized.

So just in case you missed the differences, here they are:

- `var` declarations are globally scoped or function scoped while `let` and `const` are block scoped.
- `var` variables can be updated and re-declared within its scope; `let` variables can be updated but not re-declared; `const` variables can neither be updated nor re-declared.
- They are all hoisted to the top of their scope. But while `var` variables are initialized with `undefined`, `let` and `const` variables are not initialized.
- While `var` and `let` can be declared without being initialized, `const` must be initialized during declaration.

Javascript global execution context:

JavaScript is a *synchronous* (Moves to the next line only when the execution of the current line is completed) and *single-threaded* (Executes one command at a time in a specific order one after another serially) language. To know behind the scene of how JavaScript code gets executed internally, we have to know something called **Execution Context** and its role in the execution of JavaScript code.
Execution Context: Everything in JavaScript is wrapped inside Execution Context, which is an abstract concept (can be treated as a container) that holds the whole information about the environment within which the current JavaScript code is being executed.
Now, an Execution Context has two components and JavaScript code gets executed in two phases.

- **Memory Allocation Phase:** In this phase, all the functions and variables of the JavaScript code get stored as a key-value pair inside the memory component of the execution context. In the case of a function, JavaScript copied the whole function into the memory block but in the case of variables, it assigns *undefined* as a placeholder.
- **Code Execution Phase:** In this phase, the JavaScript code is executed one line at a time inside the Code Component (also known as the Thread of execution) of Execution Context.

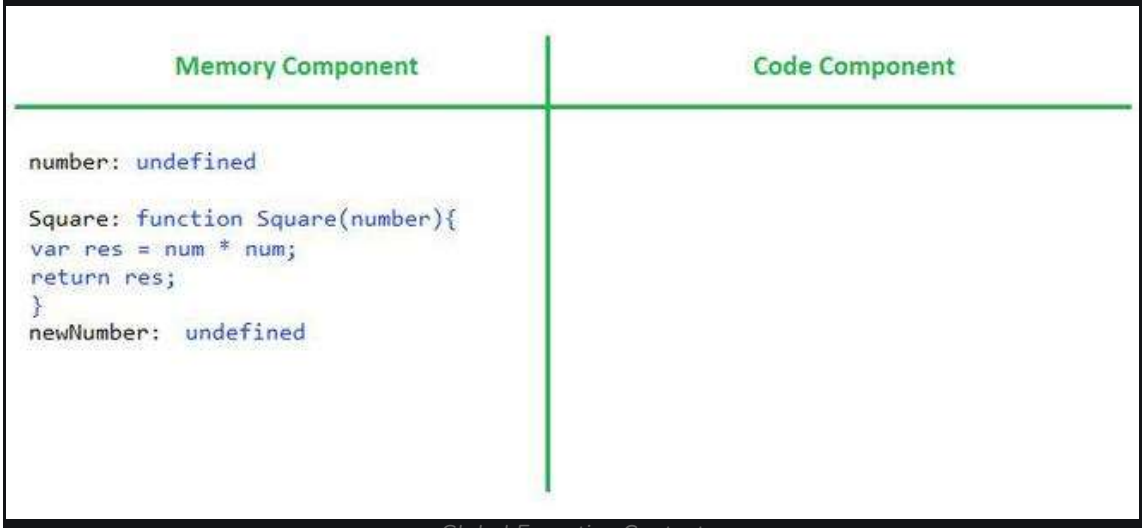
Let's see the whole process through an example.

- Javascript

```
var number = 2;
function Square (n) {
    var res = n * n;
    return res;
}
var newNumber = Square(3);
```

In the above JavaScript code, there are two variables named *number* and *newNumber* and one function named *Square* which is returning the square of the number. So when we run this program, Global Execution Context is created.

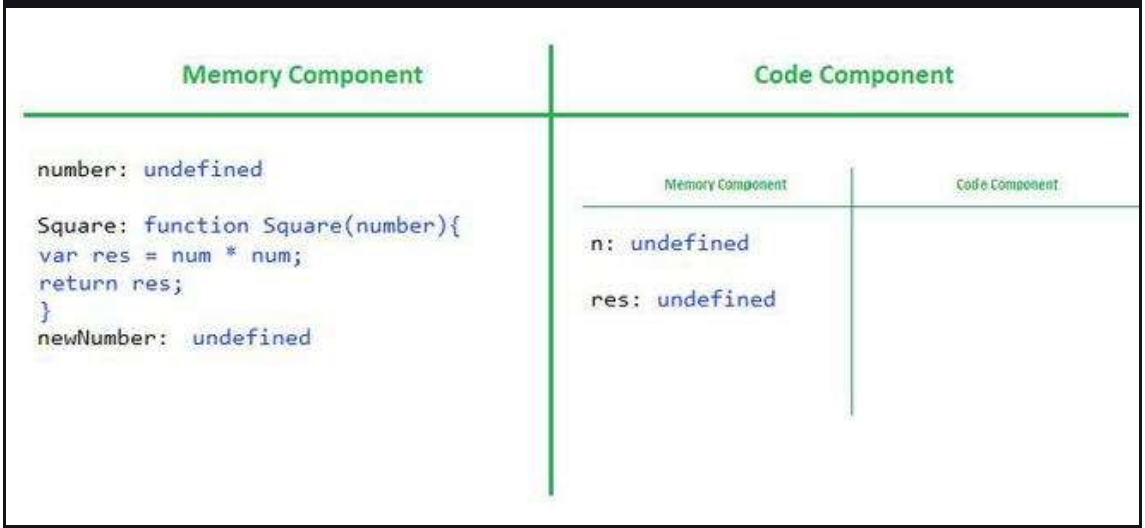
So, in the Memory Allocation phase, the memory will be allocated for these variables and functions like this.



Global Execution Context

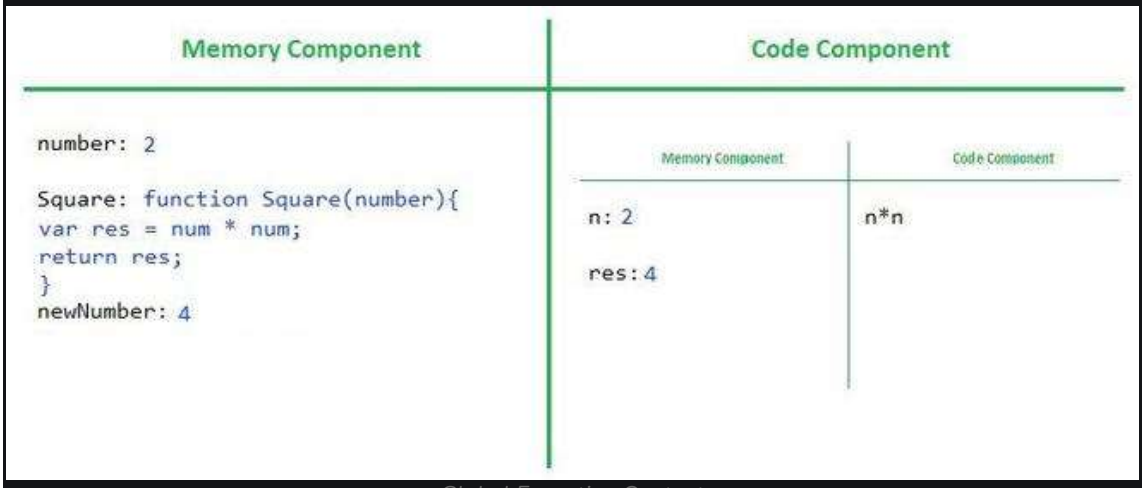
In the Code Execution Phase, JavaScript being a single thread language again runs through the code line by line and updates the values of function and variables which are stored in the Memory Allocation Phase in the Memory Component.

So in the code execution phase, whenever a new function is called, a new Execution Context is created. So, every time a function is invoked in the Code Component, a new Execution Context is created inside the previous global execution context.



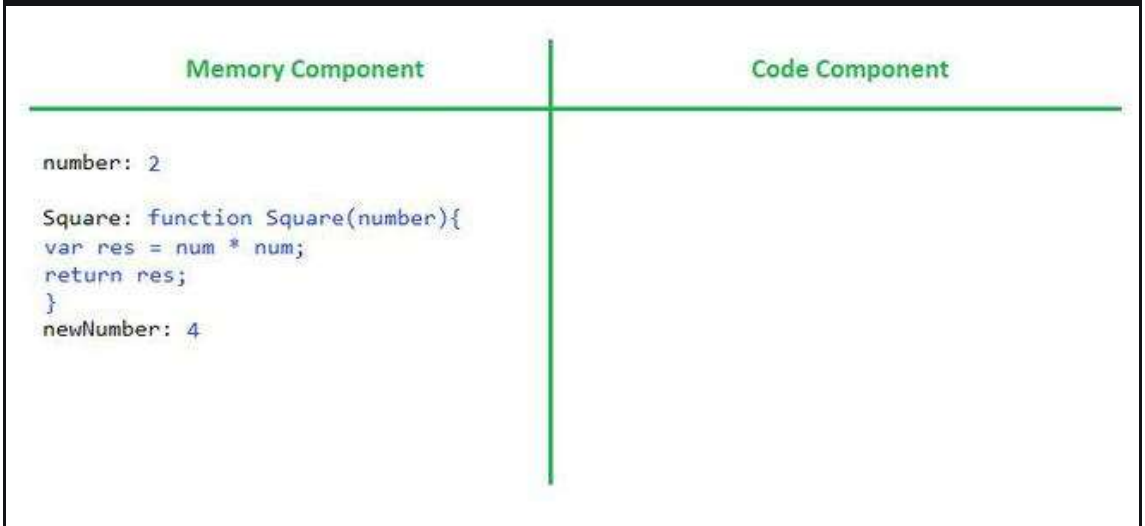
Global Execution Context

So again, before the memory allocation is completed in the Memory Component of the new Execution Context. Then, in the Code Execution Phase of the newly created Execution Context, the global Execution Context will look like the following.



Global Execution Context

As we can see, the values are assigned in the memory component after executing the code line by line, i.e. *number: 2, res: 4, newNumber: 4*. After the *return* statement of the invoked function, the returned value is assigned in place of undefined in the memory allocation of the previous execution context. After returning the value, the new execution context (temporary) gets completely deleted. Whenever the execution encounters the return statement, It gives the control back to the execution context where the function was invoked.



Global Execution Context

After executing the first function call when we call the function again, JavaScript creates again another temporary context where the same procedure repeats accordingly (memory execution and code execution). In the end, the global execution context gets deleted just like child execution contexts. The whole execution context for the instance of that function will be deleted

Call Stack: When a program starts execution JavaScript pushes the whole program as global context into a stack which is known as **Call Stack** and continues execution. Whenever JavaScript executes a new context and just follows the same process and pushes to the stack. When the context finishes, JavaScript just pops the top of the stack accordingly.

Call Stack

When JavaScript completes the execution of the entire code, the Global Execution Context gets deleted and popped out from the Call Stack making the Call stack empty.