# Q.1) Difference between var, let and const keywords in JavaScript

The [*var*](https://www.geeksforgeeks.org/javascript-var/) is the oldest keyword to declare a variable in [JavaScript](https://www.geeksforgeeks.org/introduction-to-javascript/).

The scope of the*var* keyword is the global or function scope.

It means variables defined outside the function can be accessed globally, and variables defined inside a particular function can be accessed within the function.

**Example 1:**

Variable ‘a’ is declared globally. So, the scope of the variable ‘a’ is global, and it can be accessible everywhere in the program. The output shown is in the console.

var a = 10

function f() {

    console.log(a)

}

f();

console.log(a);

Output:

10

10

**Example 2:**

The variable ‘a’ is declared inside the function. If the user tries to access it outside the function, it will display the error. Users can declare the 2 variables with the same name using the *var* keyword. Also, the user can reassign the value into the *var* variable. The output is shown in the console.

function f() {

    // It can be accessible any

    // where within this function

    var a = 10;

    console.log(a)

}

f();

// A cannot be accessible

// outside of function

console.log(a);

Output:

10

ReferenceError: a is not defined

The [*let*keyword](https://www.geeksforgeeks.org/javascript-let/) is an improved version of the [*var* keyword](https://www.geeksforgeeks.org/javascript-var/).

The scope of a *let*variable is only block scoped.

It can’t be accessible outside the particular block ({block}). Let’s see the below example.

let a = 10;

function f() {

    if (true) {

        let b = 9

        // It prints 9

        console.log(b);

    }

    // It gives error as it

    // defined in if block

    console.log(b);

}

f()

// It prints 10

console.log(a)

Output:

9

ReferenceError: b is not defined

The code returns an error because we are accessing the *let* variable outside the function block. The output is shown in the console.

**Example 2:**

Users cannot re-declare the variable defined with the *let*keyword but can update it.

let a = 10

// It is not allowed

let a = 10

// It is allowed

a = 10

Output:

Uncaught SyntaxError: Identifier 'a' has already been declared

**Example 3:**

Users can declare the variable with the same name in different blocks using the *let* keyword.

let a = 10

if (true) {

    let a = 9

    console.log(a) // It prints 9

}

console.log(a) // It prints 10

Output:

9

10

The [*const* keyword](https://www.geeksforgeeks.org/javascript-const/) has all the properties that are the same as the [*let* keyword](https://www.geeksforgeeks.org/javascript-let/), except the user cannot update it.

**Scope:**[block scoped:](https://www.geeksforgeeks.org/javascript-es2015-block-scoping/)

When users declare a *const* variable, they need to initialize it, otherwise, it returns an error.

The user cannot update the *const*variable once it is declared.

**Example 1:**

We are changing the value of the const variable so that it returns an error. The output is shown in the console.

const a = 10;

function f() {

    a = 9

    console.log(a)

}

f();

Output:

TypeError:Assignment to constant variable.

**Differences between**[**var**](https://www.geeksforgeeks.org/javascript-var/)**,**[**let**](https://www.geeksforgeeks.org/javascript-let/)**, and**[**const**](https://www.geeksforgeeks.org/javascript-const/)**:**

| **var** | **let** | **const** |
| --- | --- | --- |
| The scope of a [*var*](https://www.geeksforgeeks.org/javascript-var/)variable is **functional scope.** | The scope of a[*let*](https://www.geeksforgeeks.org/javascript-let/) variable is **block scope.** | The scope of a [*const*](https://www.geeksforgeeks.org/javascript-const/) variable is **block scope.** |
| It **can be updated and re-declared** into the scope. | It **can be updated but cannot be re-declared** into the scope. | It **cannot be updated or re-declared** into the scope. |
| It can be declared without initialization. | It can be declared without initialization. | It cannot be declared without initialization. |
| It can be accessed without initialization as its default value is “undefined”.  console.log(a);  var a = 10; | It cannot be accessed without initialization otherwise it will give ‘referenceError’.  console.log(a);  let a = 10; | It cannot be accessed without initialization, as it cannot be declared without initialization.  console.log(a);  const a = 10; |
| hoisting done, with initializing as ‘default’ value | Hoisting is done, but not initialized (this is the reason for the error when we access the let variable before declaration/initialization | Hoisting is done, but not initialized (this is the reason for the error when we access the const variable before declaration/initialization |

# Q.2) ES6 New String Methods

In [ES6](https://www.geeksforgeeks.org/introduction-to-es6/), four new methods were added to String.

These methods are like a boon for programmers when it comes to string manipulation in [JavaScript](https://www.geeksforgeeks.org/javascript-tutorial/).

In day to day programming, we often deal with strings.

The first three methods also reduce the dependency on Regular Expression RegExp for certain tasks.

Four ES6 New String Methods are described below:

1. **startsWith**

example

< script >

let str = "GeeksforGeeks";

console.log(str.startsWith("Geeks"));

// Here specified position is 5, that means

// searching will start from 'f' whose index

// in string str is 5

console.log(str.startsWith("for", 5));

console.log(str.startsWith("geeks"));

< /script>

Output:

true

true

false

1. **endsWith**

example

<script>

let str = "GeeksforGeeks";

console.log(str.endsWith("Geeks"));

// Here specified length is 8, that means

// length of str will be considered as 8

// and rest will be omitted

console.log(str.endsWith("for", 8));

console.log(str.endsWith("geeks"));

</script>

Output:

true

true

false

1. **includes**

example

<script>

let str = "GeeksforGeeks";

console.log(str.includes("eks"));

// Here search will start from index 8

// of str

console.log(str.includes("for", 8));

console.log(str.includes("geeks"));

</script>

Output:

true

false

false

1. **repeat**

**example**

<script>

let str = "GeeksforGeeks";

console.log(str.repeat(2));

let newStr = str.repeat(3);

console.log(newStr);

</script>

Output:

GeeksforGeeksGeeksforGeeks

GeeksforGeeksGeeksforGeeksGeeksforGeeks

# Q.3) Destructuring assignment

The **destructuring assignment** syntax is a JavaScript expression that makes it possible to unpack values from arrays, or properties from objects, into distinct variables.

1. **Array Destructuring**

Example

const vehicles = ['mustang', 'f-150', 'expedition'];

const [car, truck, suv] = vehicles;

console.log(`first=> ${ car } second=> ${ truck } third=> ${ suv }`)

output

first=> mustang second=> f-150 third=> expedition

If we only want the car and suv we can simply leave out the truck but keep the comma:

const vehicles = ['mustang', 'f-150', 'expedition'];

const [car,, suv] = vehicles;

console.log(`first=> ${ car } third=> ${ suv }`)

output

first=> mustang third=> expedition

let a, b, rest;

[a, b] = [10, 20];

console.log(a);

// Expected output: 10

console.log(b);

// Expected output: 20

[a, b, ...rest] = [10, 20, 30, 40, 50];

console.log(rest);

// Expected output: Array [30, 40, 50]

**2.Object Destructuring**

same as array but in object destructuring we just create an object and access the property of object using {} braces.

const hero = {

name: 'Batman',

realName: 'Bruce Wayne'

};

Const {name, realName} = hero;

console.log(name); *// => 'Batman',*

console.log(realName); *// => 'Bruce Wayn*

# Q.4) Arrow function expressions

An **arrow function expression** is a compact alternative to a traditional [function expression](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/function), with some semantic differences and deliberate limitations in usage:

ES6 fat arrow functions have a shorter syntax compared to function expressions and lexically bind this value.

Arrow functions are always anonymous and effectively turn function (arguments) { expression } into arguments fat arrow symbol expression.

* Arrow functions don't have their own [bindings](https://developer.mozilla.org/en-US/docs/Glossary/Binding) to [this](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this), [arguments](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments), or [super](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/super), and should not be used as [methods](https://developer.mozilla.org/en-US/docs/Glossary/Method).
* Arrow functions cannot be used as [constructors](https://developer.mozilla.org/en-US/docs/Glossary/Constructor). Calling them with [new](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) throws a [TypeError](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/TypeError). They also don't have access to the [new.target](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new.target) keyword.
* Arrow functions cannot use [yield](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/yield) within their body and cannot be created as generator functions.

Example

const gfg = () => {

    console.log( "Hi from GeekforGeeks!" );

}

gfg();

Q.5) **Default Parameter**

Function parameters with default values are initialized with default values if they contain no value or are undefined.

JavaScript function parameters are defined as undefined by default. However, it may be useful to set a different default value.

That is where default parameters come into play.

**Example**

If we do not pass a number as the second parameter and take the default parameter as the second parameter, it will multiply the first number with the default number, and if we pass two numbers as parameters, it will multiply the first number with the second number

function multiply(a, b = 2) {

    return a \* b;

}

let num1 = multiply(5);

console.log(num1);

let num2 = multiply(5, 8);

console.log(num2);

**Note:** if we don’t pass default parameter and use the parameter b in that condition it’s gives NaN.

Q.7) What is Rest operator in javascript ?

The operator is used to put some user-supplied values into an Array.

The text after the rest operator references the values you wish to encase inside an array.

You can only use it before the last parameter in a function definition.

We always use ‘ …’ in last parameter.

                                                                       OR

The rest parameter (...) allows a function to treat an indefinite number of arguments as an array.

Rest parameter is added in ES2015 or ES6 which improved the ability to handle parameter.

**Note:** When … is at the end of the function parameter, it is the rest parameter. It stores n number of parameters as an array. Let’s see how the rest parameter works:

Example

function sumAll(...args) {

 console.log(args)

}

// here the function can be called with any number of arguments

sumAll(1)

sunAll(1,2,3)

function khan(a,b,…c){ // if we have more than 1 parameter value than always     use rest operator in last value

 console.log(a,b,c)

 }

 Khan(1,2,45,67,89);

explanation

basically when we use … in parameter it means the variable takes n numbers of values and create an Array

output.

[1]

[1,2,3]

explanation

basically when we use … in parameter it means the variable takes n numbers of values and create an Array

output.

[1]

[1,2,3]

Q.8) What is Spread operator in javascript ?

While using rest operator we make an array of parameters , but sometimes we need to the exact opposite and extract all the values , this is where we use spread operator

The JavaScript spread operator (...) allows us to quickly copy all or part of an existing array or object into another array or object.

Example

const numbersOne = [1, 2, 3];

const numbersTwo = [4, 5, 6];

const numbersCombined = [...numbersOne, ...numbersTwo];

explanation

In upper code numberOne, numberTwo both the array are assign to numberCombined using spread Operator.

# Q.9) Number Methods used in ES6

# Numbers are one of the primitive data types in JavaScript.

# JavaScript does not have any specific data type (like int, float, long, etc) for different numbers.

# It has only one data type i.e. Number. It can be written with or without decimals.

# They are basically a 64-bit double-precision floating-point value.

# There are 6 different Number methods that are used in ES6:

**1.**[Number.isNaN()](https://www.geeksforgeeks.org/number-isnan-javascript/)**:**

It is used to check whether the passed value is **NaN**or not. If it is a *number*or *string* or *undefined,*it will give**false.**

Example

console.log(Number.isNaN(NaN)); // true

    console.log(Number.isNaN(0/0)); // true

    console.log(Number.isNaN(10)); // false

    console.log(Number.isNaN("Sarthak")); // false

    console.log(Number.isNaN("NaN")); // false

**2.**[Number.isInteger()](https://www.geeksforgeeks.org/javascript-number-isinteger-function/)**:**

It is used to determine whether the value passed is a integer or not. The number has to be only integer (negative or positive), else the method will give false.

Example

    console.log(Number.isInteger(0)) // true

    console.log(Number.isInteger(-9999)) // true

    console.log(Number.isInteger(99999)) // true

    console.log(Number.isInteger("Sarthak Delori")) // false

    console.log(Number.isInteger(undefined)) // false

**3.**[Number.isSafeInteger()](https://www.geeksforgeeks.org/javascript-number-issafeinteger/)**:**

It Is quite similar to Number.isInteger() in working, as it also determines whether the value passed is integer or not but within the range of -(2^53 – 1) to (2^53 – 1).

console.log(Number.isSafeInteger(0)) // true

    console.log(Number.isSafeInteger(Math.pow(2,53)-1)) // true

    console.log(Number.isSafeInteger(Math.pow(2,53))) // false

    console.log(Number.isSafeInteger("Sarthak Delori")) // false

    console.log(Number.isSafeInteger(undefined)) // false

**4.**[Number.isFinite()](https://www.geeksforgeeks.org/javascript-number-isfinite-function/)**:**

It determines whether the passed value is finite or not.

It basically takes a number and check if it is a 64-bit number.

If the passed value is a string or null value, it gives.

But JavaScript has a **global**isFinite() also which gives true in all the cases whether it is a number of less than or equal to 64 bits or a string (of a number like “999” and not “Sarthak”) or even a null value.

console.log("For Number.isFinite():")

    console.log(Number.isFinite(0)) // true

    // 2e64 === 2\*10^64

    console.log(Number.isFinite(2e64)) // true

    console.log(Number.isFinite(null)) // false

    console.log(Number.isFinite("Sarthak Delori")) // false

    console.log(Number.isFinite(undefined)) // false

    console.log("For global isFinite():")

    console.log(window.isFinite(null)) // true

    console.log(window.isFinite("999")) // true

    console.log(window.isFinite("Sarthak")) // false