1. Difference between var, let and const with code

**Var**

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.

**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.

Example:-

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";

**Hoisting of var**

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

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:

const 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.

2) Write a brief introduction on available data types in JavaScript?

JavaScript, like any other programming language, has its own data structures and types. JavaScript has a few data types that we have to know about in order to build programs with it. Different pieces of data can be put together to build more complex data structures.

JavaScript is a loosely typed, or dynamically typed, language. This means that a variable that’s declared with one type can be converted to another type without explicitly converting the data to another type. Variables can also contain any type at any time, depending on what’s assigned. For example, if we write the following code:

let x = 1;   
x = 'bar';  
x = true;

In the first line x is a number, but in the second line, the same variable x has been reassigned into a string. In the last line, it has been again been reassigned, this time to a Boolean.

JavaScript has multiple data types. There are seven primitive data types and an object type. The seven primitive types are Boolean, null, undefined, number, BigInt, string, and symbol.

All of JavaScript’s primitive types are immutable, which means that they can’t be changed. The primitive types contain values that are fixed once they’re defined. The Boolean type is either true or false and represents logical entities. The null type only has one value, which is null . The null value means that it refers to some nonexistent or invalid object or address. The undefined type is unique to JavaScript. It means that a variable hasn’t been assigned any value.

# Numbers

There are two number types in JavaScript, which are number and BigInt. The number type is a double-precision 64-bit number that can have values between -2 to the 53rd power minus 1 and 2 to the 53rd power minus 1. There’s no specific type for integers. All numbers are floating point numbers. There are also three symbolic values: Infinity , -Infinity and NaN.

The largest and smallest available values for a number are Infinity and -Infinity respectively. We can also use the constants Number.MAX\_VALUE or Number.MIN\_VALUE to represent the largest and smallest numbers. We can use the Number.isSafeInteger() function to check whether a number is in the range of numbers available that are allowed in JavaScript. There are also the constants Number.MAX\_SAFE\_INTEGER and Number.MIN\_SAFE\_NUMBER to check if the number you specify in the safe range. Anything outside the range isn’t safe and will be a double-precision floating point of the value. The number 0 has two representations in JavaScript: There’s +0 and -0, and 0 is an alias for +0. It will noticed if you try to divide a number by 0:

1/+0//Infinity  
1/-0 // -Infinity

Sometimes numbers can represent Boolean values with bitwise operators to operate them as Boolean, but this is bad practice since JavaScript already has Boolean types, so using numbers to represent Boolean will be unclear to people reading the code. That’s because numbers can represent numbers, or they can represent Booleans if someone chooses to use them that way.

In JavaScript, there is the BigInt type to store numbers that are beyond safe integer range. A BigInt number can be created by adding an n character to the end of a number. With BigInt, we can make calculations that have results beyond the safe range of normal numbers. For example, we can write the following expressions and still get the numbers we expect:

const x = 2n \*\* 55n;  
const y = x + 1n;

For x we get 36028797018963968n and for y we get 36028797018963969n , which is what we expect. BigInts can use the same arithmetic operations as numbers like +, \*, -, \*\* and %. A BigInt behaves like a number when converted to a Boolean with functions, keywords, or operators like Boolean, if , || , &&, !. BigInts cannot be operated in the same expressions as numbers. If we try that, we will get a TypeError.

Example:-

var x; // Now x is undefined

x = 5; // Now x is a Number

# Strings

Strings are used to represent textual data. Each element of the string has its own position in the string. It’s zero-indexed, so the position of the first character of a string is 0. The length property of the string has the total number of characters of the string.

JavaScript strings are immutable. We cannot modify a string that has been created, but we can still create a new string that contains the originally defined string. We can extract substrings from a string with the substr() function and use the concat() function to concatenate two strings.

We should only present text data with strings. If there are more complex structures needed for your data structure, then they shouldn’t be represented with a string. Instead, they should be objects. This is because it’s easy to make mistakes with strings since we can put in the characters we want. Therefore, mistakes are made easily.

Example:-

var day = "wednesday";

# Symbols

It is a unique and immutable identifier. Once you have created it, it cannot be copied. Every time you create a new symbol, it’s a unique one. Symbols are mainly used for unique identifiers in an object. There are some static properties and methods of its own that expose the global symbol registry. It is like a built-in object, but it doesn’t have a constructor, so we can’t write new Symbol to construct a symbol object with the new keyword.

To create new symbols, we can write:

const fooSymbol = Symbol('foo')

Note that each time we call the Symbol function, we get a new symbol, so if we write

Symbol('sym') === Symbol('sym')

# Objects

Object is a reference data type, which means it can be referenced by an identifier that points to the location of the object in memory. In memory, the object’s value is stored, and, with the identifier, we can access the value. Object has properties, which are key-value pairs with the values being able to contain the data with primitive types or other objects. That means we can use object to build complex data structures. The key is an identifier for the values of a property, which can be stored as a string or symbol. There are two types of properties that have certain attributes in an object. Objects have data properties and accessor properties.

A JavaScript object has the following data properties:

* [[Value]] — This can be of any type. It has the value retrieved by a getter of the property. Defaults to undefined.
* [[Writable]] — This is a Boolean value. If it’s false, then [[Value]] can’t be changed. Defaults to false.
* [[Enumerable]] — This is a Boolean value. If it’s true, then it can be iterated over by the for...in loop, which is used to iterate over the properties of an object. Defaults to false.
* [[Configurable]] — This is a Boolean value. If it’s true, then the property can be deleted or changed to an accessor property, and all attributes can be changed. Otherwise, the property can’t be deleted or changed to an accessor property and attributes other than [[Value]] and [[Writable]] can’t be changed. Defaults to false.

Example:-

var person = {firstName:"megha", lastName:"bai", age:21};

3) Explore and Explain the various method in console function with example?

1. **console.log()**

Mainly used to log(print) the output to the console. We can put any type inside the log(), be it a string, array, object, boolean etc.

Example:-

Console.log(‘abc’);

Console.log(true);

1. **console.error()**

Used to log error message to the console. Useful in testing of code. By default the error message will be highlighted with red color.

Example:-

|  |  |  |
| --- | --- | --- |
| console.error('This is a simple error');   1. **console.warn()**   Used to log warning message to the console. By default the warning message will be highlighted with yellow color.  Example:-   |  |  | | --- | --- | | console.warn('This is a warning.');     1. **console.clear()**   Used to clear the console. The console will be cleared, in case of Chrome a simple overlayed text will be printed like : ‘Console was cleared’ while in firefox no message is returned.  Example:-   |  | | --- | | console.clear(); |  1. **console.table()**   This method allows us to generate a table inside a console. The input must be an array or an object which will be shown as a table.  Example:-  console.table({'a':1, 'b':2}); | |

1. **console.time() and console.timeEnd()**

Whenever we want to know the amount of time spend by a block or a function, we can make use of the time() and timeEnd() methods provided by the javascript console object. They take a label which must be same, and the code inside can be anything( function, object, simple console).

Example:-

|  |  |  |
| --- | --- | --- |
| console.time('scriptj');   let fun =  function(){       console.log('fun is running');   }   let fun2 = function(){       console.log('fun2 is running..');   }   fun(); // calling fun();   fun2(); // calling fun2();  console.timeEnd('scriptj');   1. **console.count()**   This method is used to count the number that the function hit by this counting method.   |  |  | | --- | --- | | Example:-  for(let i=0;i<5;i++){      console.count(i);  }   1. **console.group() and console.groupEnd()**   group() and groupEnd() methods of the console object allows us to group contents in a separate block, which will be indented.  Example:-   |  | | --- | | console.group('simple');  console.groupEnd('simple'); | | |

|  |
| --- |
|  |