JavaScript 1

1. Name the three ways to declare a variable?

You can declare a variable in JavaScript in 3 ways and these are as follows:

var: This keyword is used to declare variable globally. If you used this keyword to declare variable then the variable can accessible globally and changeable also.

```
<script>
    var myVariable = “variable-value”;
</script>
```

let: This keyword is used to declare variable locally. If you used this keyword to declare variable then the variable can accessible locally and it is changeable as well. It is good if the code gets huge.

```
<script>
    let myVariable = “variable-value”;
</script>
```

const: This keyword is used to declare variable locally. If you use this keyword to declare a variable then the variable will only be accessible within that block similar to the variable defined by using let and difference between let and const is that the variables declared using const values can’t be reassigned. So we should assign the value while declaring the variable.

```
<script>
    const myVariable = “variable-value”;
</script>
```

2. Which of the three variable declarations should you avoid and why?

You should avoid variables declared with the ***var*** keyword because these variables have a global or functional scope, and they do not support block-level scope. This means that if a variable is defined in a loop or in an if statement, it can be accessed outside the block and accidentally redefined, resulting in an error-ridden program. As a general rule, you should avoid using the ***var*** keyword.

3. What rules should you follow when naming variables?

JavaScript has only a few rules for variable names:

- The first character must be a letter or an underscore (_) or a dollar sign ($). You can't use a number as the first character or hyphens.

```
<script>
    let myVar = “I am a variable”;
    let _myVar = “I am a variable too”;
    let $myVar = “I am a variable using a dollar sign”;
    let 1myVar; // Cannot start with a number
    let my-var; // Cannot use hyphens
</script>
```

- The rest of the variable name can include any letter, any number, or the underscore. You can't use any other characters, including spaces, symbols, and punctuation marks.

- As with the rest of JavaScript, variable names are case sensitive. That is, a variable named `Interest_Rate` is treated as an entirely different variable than one named `interest_rate`.

- There's no limit to the length of the variable name.

- You can't use one of JavaScript's *reserved words* as a variable name. All programming languages have a supply of words that are used internally by the language and that can't be used for variable names because doing so would cause confusion (or worse).

```
<script>
    let var; // Cannot use this name, you will get an error!
</script>
```

4. What should you look out for when using the + operator with numbers and strings?

The **+** operator is used to add numbers, and is also used to concatenate numbers and strings. In addition, if you use the **+** operator to add a number with a string, you will get a string.

```
<script>
    let sum = 5 + 5; // Result 10
    let myConcatenatedVar = “Hi guys! I have ” + 2 + “ cats.”;
    console.log(myConcatenatedVar); // “Hi guys! I have 2 cats.”
    let newSum = 5 + “5”; // You will get “10”
</script>
```

5. How does the % operator work?

Modulus operator returns remainder of two operands.

```
<script>
    let a = 5;
    let b = a % 2; // Returns division remainder 1.
</script>
```

6. Explain the difference between == and ===.

JavaScript provides comparison operators that compare two operands and return a boolean value `true` or `false`.

== Operator	=== Operator
Compares the equality of two operands without considering type.	Compares equality of two operands with type.

```
<script>
    let c = 12;
    let d = “12”;
    c == d; // Returns true.
    c === d; // Returns false, due the variable c is a number and d is a string.
</script>
```

7. When would you receive a NaN result?

The NaN is an error value in JavaScript. However, technically it is the property of the global object. **You get NaN when the value cannot be computed** or as a result of attempted number coercion (type conversion) of non-numeric value (such that undefined) for which primitive numeric value is not available.

```
<script>
    let a = 5 + undefined; // Returns a NaN.
</script>
```

8. How do you increment and decrement a number?

You can do that using the following operators:

++ Operator	-- Operator
Increment operator. Increase operand value by one.	Decrement operator. Decrease value by one.

9. Explain the difference between prefixing and post-fixing increment/decrement operators.

The main different is as follows:

Prefixing	Post-fixing
Syntax: ++counter or --counter	**Syntax:** counter++ or counter--
The prefix returns the value after the increment/decrement.	The postfix returns the original value of the variable, before the increment/decrement.

10. What is operator precedence and how is it handled in JS?

Operator precedence refers to the priority given to operators while parsing a statement that has more than one operator performing operations in it. It is important to ensure the correct result and also to help the compiler understand what the order of operations should be. Operators with higher priorities are resolved first. But as one goes down the list, the priority decreases and hence their resolution.

11. How do you log information to the console?

log() is a function in JavaScript which is used to print any kind of variables defined before in it or to just print any message that needs to be displayed to the user. Syntax: **console. log(myVar);**

```
<script>
    let message = “Hello world!”;
    console.log(message); // Expected output: Hello world!
</script>
```

12. What does unary plus operator do to string representations of integers?

The unary plus (+) operator is the fastest (and preferred) method of converting something into a number. It can convert:

- string representations of integers (decimal or hexadecimal) and floats.
- booleans: true, false.

- null
- Values that can't be converted will evaluate to NaN.

```
<script>
    let a = 1;
    let b = -1;
    console.log(+a); // Expected output: 1
    console.log(+b); // Expected output: -1
    console.log(+’’); // Expected output: 0
    console.log(+true); // Expected output: 1
    console.log(+false); // Expected output: 0
    console.log(+hello); // Expected output: NaN
</script>
```

13. What are the eight data types in JavaScript?

- Boolean type
- Null type
- Undefined type
- Number type
- BigInt type
- String type
- Symbol type
- Objects

14. Which data type is NOT primitive?

The **object** is a non-primitive data type in JavaScript. Arrays and Functions in JavaScript belong to the **object** data type.

15. What is the relationship between null and undefined?

In JavaScript, **undefined** is a type, whereas **null** an object.

Undefined	Null
It means a variable declared, but not value has been assigned a value. e.g. let myVar;	It is an assignment value, so, you can assign it to a variable. e.g. let myVar = null;

```
<script>
    let a;
    let b = null;
    console.log(a); // Expected output: undefined
    console.log(typeof(a)); // Expected output: undefined
    console.log(b); // Expected output: null
    console.log(typeof(b)); // Expected output: object
</script>
```

16. What is the difference between single, double, and backtick quotes for strings?

There is no real difference between using single quotes, double quotes, or backticks. You can choose one or multiple styles based on your preference. However, it is always good to stick to a single format throughout the project to keep it neat and consistent.

In JavaScript, single quotes (‘’) and double quotes (“”) are used to create string literals. Most developers use single or double quotes as they prefer, and sometimes they let their code formatters decide what to use.

```
<script>
    ‘myString’ === “myString”
    // Whichever quoting style you open a string with, close it with the same.
    ‘myString’ // Correct
    “myString” // Correct
    ‘myString” // Incorrect
</script>
```

The only noticeable difference between single quotes and double quotes comes into play when we have to escape characters.

If you use single quotes to create a string, you cannot use single quotes within that string without escaping them using a backslash (\).

```
<script>
    let myString1 = ‘I’m a cat’; // Incorrect
    let myString2 = ‘I\’m a cat’;
    Console.log(myString2); // Expected output: I’m a cat
</script>
```

The same theory applies to double quotes, and you have to use a backslash to escape any double quotes inside double quotes.

```
<script>
    let myString1 = “I’m a cat”; // Incorrect
    let myString2 = “Hello, my name is \“Emily\””;
    console.log(myString2); // Expected output: Hello, my name is “Emily”
</script>
```

However you can use single quotes inside double quotes or double quotes inside single quotes without escaping.

```
<script>
    let myString1 = “I’m a cat”; // Correct
    let myString2 = ‘Hello, my name is “Emily”’; // Correct too
    Console.log(myString1); // Expected output: I’m a cat
    console.log(myString2); // Expected output: Hello, my name is “Emily”
</script>
```

Backticks are an ES6 feature that allows you to create strings in JavaScript. Although backticks are mostly used for HTML or code embedding purposes, they also act similar to single and double quotes. Besides, using backticks makes it easier for string operations.

```
<script>
    let myAge = 13;
    // Expected output: My name is Emily and I’m 13 years old.
    console.log(“My name is Emily and I’m ” + myAge + “ years old.”);
    // Expected output: My name is Emily and I’m 13 years old.
    console.log(`My name is Emily and I’m ${myAge} years old.`);
</script>
```

17. What is the term for embedding variables/expressions in a string?

JavaScript string interpolation is the process of embedding an expression into part of a string. A template literal is used to embed expressions. You can add values such as variables and mathematical calculations into a string using interpolation.

18. Which type of quote lets you embed variables/expressions in a string?

You should use **backticks** for this task. e.g.

```
<script>
    let myCats = 2;
    Console.log(`I have ${myCats} beautiful cats`);
    // Expected output: I have 2 beautiful cats
</script>
```

19. How do you embed variables/expressions in a string?

One special feature of the template literal feature is the ability to include expressions and variables within a string. Instead of having to use concatenation, we can **use the ${} syntax to insert a variable**.

20. How do you escape characters in a string?

We can use the backslash (\) escape character to prevent JavaScript from interpreting a quote as the end of the string.
The syntax of \' will always be a single quote, and the syntax of \" will always be a double quote, without any fear of breaking the string.

21. What is the difference between the slice/substring/substr string methods?

The methods **slice(), substring(),** and **substr()** are the methods that extract parts of a string and then return the extracted parts in a new string. Also, all of these methods do not change the original string from which they extract.

substring()
The `substring()` function is the most common way to get a substring in JavaScript. It takes two parameters: `indexStart` and `indexEnd`. It returns the portion of the string that starts at `indexStart` and ends the character immediately preceding `indexEnd`. For example:

```
<script>
    const str = “Twas the night before Christmas”;
    let startIndex = 0;
    let endIndex = 4;
    str.substring(startIndex, endIndex); // Result: ‘Twas’
    str.substring(5, 14); //Result: ‘the night’
    // If you don’t specify indexEnd
    str.substring(5) // Result: ‘the night before Christmas’
    // str.substring(4, -1) is equivalent to str.substring(4, 0)
    str.substring(4, -1) // Result: ‘Twas’
</script>
```

The **substring()** function has some quirky behavior in edge cases:
- If `indexStart` or `indexEnd` is less than 0, it is treated as 0.
- If `indexEnd` < `indexStart`, the two are swapped.

substr()

The `substr()` function is also common, but it is considered a *"legacy function"* in **Mozilla's docs**. You shouldn't use it when writing new code, but you may see it in existing JavaScript projects. The key difference between `substring()` and `substr()` is that `substr()` has a different 2nd parameter. The first parameter to substr() is `start`, and the 2nd is `length`. For example:

```
<script>
    const str = “Twas the night before Christmas”;
    let start = 0;
    let length = 4;
    // if `start === 0`, substring() and substr() are similars
    str.substr(start, length); // Result: ‘Twas’
    str.substring(5, 9); //Result: ‘the night’
    ‘the night’.length; // Result: 9
</script>
```

Unlike `substring()`, you can call `substr()` with a negative `start`. That will make `substr()` start counting at the end of the string as opposed to the beginning. For example:

```
<script>
    const str = “Twas the night before Christmas”;
    let start = -9;
    let length = 9;
    str.substr(start, length); // Result: ‘christmas’
    ‘christmas’.length; // Result: 9
</script>
```

slice()

The `slice()` function is less common than `substring()` and `substr()`. However, it has the best aspects of both `substring()` and `substr()`. Like `substring()`, the `slice()` function takes the start and end indices as parameters, and is not considered a legacy function. Like `substr()`, the `slice()` function supports negative indices. For example:

```
<script>
    const str = “Twas the night before Christmas”;
    str.slice(0, 4); // Result: ‘Twas’
    str.slice(5, 14); // Result: ‘the night’
    str.slice(-16, -10); // Result: ‘before’
    str.slice(-9); // Result: ‘christmas’
</script>
```

22. What are the three logical operators and what do they stand for?

JavaScript provides three logical operators:

- `!` (Logical NOT): The operator accepts a single argument and does the following:
  - Converts the operand to boolean type: true/false.
  - Returns the inverse value.

```
<script>
    console.log(!true); // Output: false
    console.log(!0); // Output: true
</script>
```

- `||` (Logical OR): In classical programming, the logical OR is meant to manipulate boolean values only. If any of its arguments are `true`, it returns `true`, otherwise it returns `false`.

```
<script>
    console.log(true || true); // Output: true
    console.log(false || true); // Output: true
    console.log(true || false); // Output: true
    console.log(false || false); // Output: false
</script>
```

- `&&` (Logical AND): In classical programming, AND returns true if both operands are `truthy` and `false` otherwise:

```
<script>
    console.log(true && true); // Output: true
    console.log(false && true); // Output: false
    console.log(true && false); // Output: false
    console.log(false && false); // Output: false
</script>
```

23. What are the comparison operators?

Comparison operators are used in logical statements to determine equality or difference between variables or values.

Operator	Description
==	Equal to
===	Equal value and equal type
!=	Not equal
!==	Not equal value or not equal type
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

24. What are truthy and falsy values?

JavaScript uses type coercion (implicit conversion of values from one data type to another) in Boolean contexts, such as conditionals. This means that values are considered either **truthy** (evaluate to `true`) or **falsy** (evaluate to `false`) depending on how they are evaluated in a Boolean context.

25. What are the falsy values in JavaScript?

There are 6 values that are considered **falsy** in JavaScript:

- The keyword `false`
- The primitive value `undefined`
- The primitive value `null`
- The empty string (`'', ""`)
- The global property `NaN`
- A number or BigInt representing `0 (0, -0, 0.0, -0.0, 0n)`

26. What are conditionals?

Conditional statements control behaviour in JavaScript and determine whether or not pieces of code can run. There are multiple different types of conditionals in JavaScript including:

- **If statements**: where if a condition is true it is used to specify execution for a block of code.
- **Else statements**: where if the same condition is false it specifies the execution for a block of code.
- **Else if statements**: this specifies a new test if the first condition is false.
- **Switch statements**: to specify many alternative blocks of code to be executed

27. What is the syntax for an if/else conditional?

`if` and `else` statements have the following syntax

```
<script>
    if (condition) {
       // Block of code to be executed if the condition is true
    } else {
       // Block of code to be executed if the condition is false
    }
</script>
```

28. What is the syntax for a switch statement?

`switch` statement has the following syntax:

```
<script>
    switch (expression) {
       case a:
          // Code block
          break;
       case b:
          // Code block
          break;
       Default:
          // Code block
     }
</script>
```

29. What is the syntax for a ternary operator?

The **conditional (ternary) operator** is the only JavaScript operator that takes three operands: a condition followed by a question mark `(?)`, then an expression to execute if the condition is ***truthy*** followed by a colon `(:)`, and finally the expression to execute if the condition is ***falsy***. This operator is frequently used as an alternative to an `if...else` statement.

```
<script>
    condition ? exprIsTrue : exprIsFalse
</script>
```

30. What is nesting?

Nesting is when you write something inside of something else. e.g.

```
if (daylight) {
    if (before 12) {
        // It's morning
    } else {
        // It's afternoon
    }
}
```

You can have a function inside of another function: `function x () { function y() { // something; } }`

31. What are functions useful for?

A **function** is a block of reusable code written to perform a specific task. You can think of a function as a sub-program within the main program. A function consists of a set of statements but executes as a single unit.

```
function sayHello() {
    console.log(“Hello! I’m a function.”);
}
```

32. How do you invoke a function?

You can call a function in the following way:

```
function sayHello() {
    console.log(“Hello! I’m a function.”);
}
sayHello(); // Output: Hello I’m a function.
```

You should use the `functionName();` to invoke it.

33. What are anonymous functions?

Anonymous Function is a function that does not have any name associated with it. Normally we use the *function* keyword before the function name to define a function in JavaScript, however, in anonymous functions in JavaScript, we use only the *function* keyword without the function name.

```
// Anonymous function example
let firstMessage = function() {
         console.log(“I’m an anonymous function”)
};
firstMessage(); // Output: I’m an anonymous function

// Arrow function example
let secondMessage = () => console.log(“I’m an arrow function”);
secondMessage(); // Output: I’m an arrow function
```

34. What is function scope?

Function Scope: When a variable is declared inside a function, it is only accessible within that function and cannot be used outside that function.

35. What are return values?

The **return** statement is ***used to return a particular value from the function to the function caller***. The function will stop executing when the return statement is called. The return statement should be the last statement in a function because the code after the return statement will be unreachable.

36. What are arrow functions?

Arrow functions are a new way to write anonymous function expressions. Arrow functions differ from traditional functions in a number of ways, including the way their scope is determined and how their syntax is expressed. Because of this, arrow functions are particularly useful when passing a function as a parameter to a higher-order function, such as when you are looping over an *array* with *built-in iterator methods*. Their syntactic abbreviation can also allow you to improve the readability of your code. e.g. `hello = () => "Hello World!";`