what is java script?

JavaScript is a programming language that is primarily used for creating interactive web pages and web applications. It is a high-level, interpreted language that was initially developed by Brendan Eich in 1995 for use in web browsers. Since then, it has become one of the most widely used programming languages in the world, both for web development and other applications.

JavaScript is a dynamically typed language, meaning that the type of a variable is determined at runtime rather than at compile time. It is also a loosely typed language, meaning that variables can be assigned values of different types without the need for explicit type conversion.

JavaScript code is typically embedded in HTML files and executed by web browsers. However, it can also be used for server-side programming with the help of tools like Node.js. Some common uses for JavaScript include:

- Adding interactive features to websites, such as form validation, dropdown menus, and slideshows
- Creating web applications, such as social media platforms and online marketplaces
- Building mobile applications using frameworks like React Native
- Developing desktop applications using tools like Electron

JavaScript has a number of powerful features, including support for object-oriented programming, functional programming, and asynchronous programming. Its popularity and versatility have made it a key skill for developers across a wide range of industries.

Data type in java Script

1.Primitive Data type
2.Non-Primitive Data type

JavaScript has several data types, which can be categorised as follows:

1. Primitive Data Types:
    - Number: represents both integer and floating-point numbers
    - String: represents textual data
    - Boolean: represents a logical value, either true or false
    - Undefined: represents a variable that has not been assigned a value
    - Null: represents a deliberate non-value, indicating the absence of any object value
2. Non-primitive Data Types:
    - Object: represents a collection of key-value pairs, where the keys are strings and the values can be any data type, including other objects
    - Array: a specialised type of object that represents a collection of values, where each value is assigned a numeric index
    - Function: a specialised type of object that represents a callable unit of code, which can be invoked with arguments and may return a value
3. Special Data Types:
    - Symbol: represents a unique identifier, used primarily for creating object properties that are not accessible via standard iteration methods.

In JavaScript, variables are dynamically typed, meaning that the data type of a variable is not explicitly declared and can change during runtime. This allows for greater flexibility but also requires careful attention to data types to avoid unexpected results. Additionally, JavaScript also supports type coercion, which is the automatic conversion of a value from one data type to another.

1.String

In JavaScript, a `String` is a sequence of characters that represent text. Strings are used extensively in JavaScript for things like storing user input, manipulating text, and generating output.

You can create a string value using string literals or by assigning a string value to a variable. For example:

javascript

```
let greeting = "Hello, world!"; // a string literal
let name = "Alice"; // a string literal
let message = "Hello, " + name + "!"; // concatenating strings with the + operator
```

JavaScript provides a number of built-in methods for working with strings, including:

- `charAt()`: returns the character at a specified index
- `concat()`: concatenates two or more strings
- `indexOf()`: returns the index of the first occurrence of a specified substring
- `lastIndexOf()`: returns the index of the last occurrence of a specified substring
- `slice()`: extracts a section of a string and returns it as a new string
- `split()`: splits a string into an array of substrings based on a specified separator
- `toLowerCase()`: converts all characters in a string to lowercase
- `toUpperCase()`: converts all characters in a string to uppercase
- `trim()`: removes whitespace from both ends of a string

Here is an example of using some of these methods:

```
let myString = " Hello, world! ";
console.log(myString.charAt(0)); // " "
console.log(myString.concat(" Welcome!")); // " Hello, world! Welcome!"
console.log(myString.indexOf("world")); // 8 console.log(myString.lastIndexOf("o")); // 10
console.log(myString.slice(7, 12)); // "world"
console.log(myString.split(" ")); // ["", "", "", "Hello,", "world!", "", "", ""]
console.log(myString.toLowerCase()); // " hello, world! " console.log(myString.toUpperCase()); // " HELLO,
WORLD! " console.log(myString.trim()); // "Hello, world!"
```

Note that most string methods return a new string rather than modifying the original string in place. This is because strings are immutable in JavaScript, meaning their values cannot be changed once they are created.

Number 👍

In JavaScript, the `Number` data type represents both integer and floating-point numbers. Numbers in JavaScript are stored as 64-bit floating-point values, which means they can represent a wide range of numbers, from very small values (e.g. 5.3 x 10^-324) to very large values (e.g. 1.8 x 10^308).

JavaScript provides a number of built-in methods for working with numbers, including:

- `toFixed()`: returns a string representation of the number with a fixed number of decimal places
- `toPrecision()`: returns a string representation of the number with a specified number of significant digits
- `toExponential()`: returns a string representation of the number in exponential notation with a specified number of decimal places
- `parseInt()`: parses a string and returns an integer
- `parseFloat()`: parses a string and returns a floating-point number
- `isNaN()`: returns true if the value is NaN (Not a Number)
- `isFinite()`: returns true if the value is a finite number, i.e. not NaN, Infinity, or -Infinity
- `Math.abs()`: returns the absolute value of a number
- `Math.ceil()`: returns the smallest integer greater than or equal to a number
- `Math.floor()`: returns the largest integer less than or equal to a number

- `Math.round()`: rounds a number to the nearest integer
- `Math.max()`: returns the largest of two or more numbers
- `Math.min()`: returns the smallest of two or more numbers
- `Math.pow()`: returns the value of a number raised to a specified power
- `Math.sqrt()`: returns the square root of a number

Here is an example of using some of these methods:

```
let myNumber = 3.14159;
console.log(myNumber.toFixed(2)); // "3.14"
console.log(myNumber.toPrecision(3)); // "3.14"
console.log(myNumber.toExponential(3)); // "3.142e+0"
console.log(parseInt("42")); // 42
console.log(parseFloat("3.14")); // 3.14
console.log(isNaN(NaN)); // true
console.log(isFinite(Infinity)); // false
console.log(Math.abs(-42)); // 42
console.log(Math.ceil(3.14)); // 4
console.log(Math.floor(3.14)); // 3
console.log(Math.round(3.14)); // 3
console.log(Math.max(3, 5, 7)); // 7
console.log(Math.min(3, 5, 7)); // 3
console.log(Math.pow(2, 3)); // 8
console.log(Math.sqrt(16)); // 4
```

Boolean 👍
In JavaScript, the `Boolean` data type represents a value that is either `true` or `false`. There are several ways to create a `Boolean` value in JavaScript:

- Using the `Boolean()` constructor: `let myBool = new Boolean(true);`
- Using the `true` or `false` keywords: `let myBool = true;`
- Using a comparison operator, which returns a `Boolean` value: `let myBool = (3 > 2);`

JavaScript also provides several built-in methods for working with `Boolean` values:

- `toString()`: returns a string representation of the `Boolean` value
- `valueOf()`: returns the primitive value of the `Boolean` object

Here is an example of using these methods:

```
let myBool = true;
console.log(myBool.toString()); // "true"
console.log(myBool.valueOf()); // true
```

It's worth noting that `Boolean` values are automatically converted to `true` or `false` in certain situations. For example, when used in an `if` statement, a `Boolean` value is converted to `true` if it is not `null`, `undefined`, `NaN`, `0`, `""` (an empty string), or `false`.

Null :

In JavaScript, `null` is a special value that represents the intentional absence of any object value. It is often used to indicate that a variable or property does not have a value or that a function does not return a value.

Unlike other primitive values in JavaScript, `null` is not a type. Instead, it is a special value of the `object` type. This is why the `typeof` operator returns `'object'` when applied to `null`.

In JavaScript, an object is a collection of properties, where each property has a name and a value. Properties can be added, modified, or deleted from an object at any time.

There are several ways to create an object in JavaScript:

- Object literals: `{}` or `let obj = {key1: value1, key2: value2}`
- Constructor function: `function Person(name) { this.name = name; }` and creating an instance with `let person = new Person('John');`
- Object.create(): `let obj = Object.create(null);`

JavaScript objects have several built-in methods and properties that can be used to interact with them:

- `Object.keys(obj)`: returns an array of the object's own enumerable property names
- `Object.values(obj)`: returns an array of the object's own enumerable property values
- `Object.entries(obj)`: returns an array of the object's own enumerable property key-value pairs as arrays
- `Object.getOwnPropertyNames(obj)`: returns an array of the object's own property names, regardless of whether they are enumerable or not
- `Object.getOwnPropertyDescriptors(obj)`: returns an object containing the descriptors for all of the object's own properties
- `Object.getOwnPropertyDescriptor(obj, prop)`: returns the descriptor for a specific property of the object
- `Object.defineProperty(obj, prop, descriptor)`: defines a new property on the object or modifies an existing property with the given descriptor
- `Object.defineProperties(obj, descriptors)`: defines multiple new properties on the object or modifies existing properties with the given descriptors
- `Object.freeze(obj)`: freezes the object, preventing any further changes to its properties
- `Object.seal(obj)`: seals the object, preventing the addition of new properties and marking all existing properties as non-configurable
- `Object.isFrozen(obj)`: returns a Boolean indicating whether the object is frozen
- `Object.isSealed(obj)`: returns a Boolean indicating whether the object is sealed
- `Object.isExtensible(obj)`: returns a Boolean indicating whether new properties can be added to the object

Here's an example of using some of these methods:

```
let person = {
    name: "John",
    age: 30,
    city: "New York"
};

console.log(Object.keys(person)); // ["name", "age", "city"]
console.log(Object.values(person)); // ["John", 30, "New York"]
console.log(Object.entries(person)); // [["name", "John"], ["age", 30], ["city", "New York"]]

let descriptor = Object.getOwnPropertyDescriptor(person, "name");
console.log(descriptor); // { value: "John", writable: true, enumerable: true, configurable: true }

Object.defineProperty(person, "name", { writable: false });
person.name = "Peter"; // Error: Cannot assign to read only property 'name' of object
```

Array 👍

JavaScript arrays have several built-in methods and properties that can be used to interact with them:

Properties:

- `Array.length`: returns the number of elements in the array

Methods:

- `Array.concat(array1, array2, ..., arrayN)`: returns a new array that is the result of concatenating two or more arrays

- `Array.join(separator)`: returns a string that is the result of concatenating all elements of the array with a separator
- `Array.pop()`: removes and returns the last element of the array
- `Array.push(element1, ..., elementN)`: adds one or more elements to the end of the array and returns the new length of the array
- `Array.shift()`: removes and returns the first element of the array
- `Array.unshift(element1, ..., elementN)`: adds one or more elements to the beginning of the array and returns the new length of the array
- `Array.slice(start, end)`: returns a shallow copy of a portion of the array, specified by the start and end indices
- `Array.splice(start, deleteCount, item1, item2, ..., itemN)`: removes elements from the array and optionally inserts new elements at the same position
- `Array.sort(compareFunction)`: sorts the elements of the array in place, using an optional compare function to determine the sort order
- `Array.reverse()`: reverses the order of the elements in the array in place
- `Array.indexOf(searchElement, fromIndex)`: returns the index of the first occurrence of a specified element in the array, starting from a specified index
- `Array.lastIndexOf(searchElement, fromIndex)`: returns the index of the last occurrence of a specified element in the array, starting from a specified index
- `Array.find(callback(element[, index[, array]]))`: returns the value of the first element in the array that satisfies the provided testing function
- `Array.findIndex(callback(element[, index[, array]]))`: returns the index of the first element in the array that satisfies the provided testing function
- `Array.forEach(callback(currentValue[, index[, array]])[, thisArg])`: executes a provided function once for each array element
- `Array.map(callback(currentValue[, index[, array]])[, thisArg])`: creates a new array with the results of calling a provided function on every element in the calling array
- `Array.filter(callback(element[, index[, array]])[, thisArg])`: creates a new array with all elements that pass the test implemented by the provided function
- `Array.reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])`: applies a function against an accumulator and each element in the array (from left to right) to reduce it to a single value
- `Array.some(callback(currentValue[, index[, array]])[, thisArg])`: tests whether at least one element in the array passes the test implemented by the provided function
- `Array.every(callback(currentValue[, index[, array]])[, thisArg])`: tests whether all elements in the array pass the test implemented by the provided function
- `Array.fill(value[, start[, end]])`: fills all the elements of the array from a start index to an end index with a static value
- `Array.includes(valueToFind[, fromIndex])`: determines whether an array includes a certain element, returning true or false as appropriate
- `Array.flat([depth])`: creates a new array with all sub-array elements concatenated into it recursively up to the specified depth

- `Array.flatMap(callback(currentValue[, index[, array]])[, thisArg])`: creates a new array with the results of calling a provided function on every element in the calling array, and then flattening the result by one level

```
// Creating an array
let fruits = ["apple", "banana", "orange"];

// Adding elements to the end of the array
fruits.push("grape", "mango");
console.log(fruits); // ["apple", "banana", "orange", "grape", "mango"]

// Removing the last element from the array
let lastFruit = fruits.pop();
console.log(lastFruit); // "mango"
console.log(fruits); // ["apple", "banana", "orange", "grape"]

// Adding elements to the beginning of the array
fruits.unshift("pear", "kiwi");
console.log(fruits); // ["pear", "kiwi", "apple", "banana", "orange", "grape"]

// Removing the first element from the array
let firstFruit = fruits.shift();
console.log(firstFruit); // "pear"
console.log(fruits); // ["kiwi", "apple", "banana", "orange", "grape"]

// Accessing a portion of the array
let citrus = fruits.slice(2, 4);
console.log(citrus); // ["banana", "orange"]

// Removing elements from the array and inserting new ones
let removedFruits = fruits.splice(1, 2, "pineapple", "watermelon");
console.log(removedFruits); // ["apple", "banana"]
console.log(fruits); // ["kiwi", "pineapple", "watermelon", "orange", "grape"]

// Sorting the array
fruits.sort();
console.log(fruits); // ["grape", "kiwi", "orange", "pineapple", "watermelon"]

// Reversing the order of the array
fruits.reverse();
console.log(fruits); // ["watermelon", "pineapple", "orange", "kiwi", "grape"]

// Filtering elements of the array based on a condition
let longFruits = fruits.filter(function(fruit) {
  return fruit.length > 5;
});
console.log(longFruits); // ["watermelon", "pineapple"]

// Mapping the elements of the array to a new array with modified values
let uppercasedFruits = fruits.map(function(fruit) {
  return fruit.toUpperCase();
});
console.log(uppercasedFruits); // ["WATERMELON", "PINEAPPLE", "ORANGE", "KIWI", "GRAPE"]
```

In JavaScript, functions are objects that can be invoked to perform a specific task. There are several types of functions in JavaScript, including:

1. Function Declarations: Function declarations are defined using the `function` keyword followed by the function name, parameters in parentheses, and function body in curly braces. These functions can be invoked using their name.

Javascript

```
function addNumbers(a, b) {
    return a + b;
  }
  console.log(addNumbers(2, 3)); // Output: 5
```

2.Function Expressions: Function expressions are defined by assigning an anonymous function to a variable. These functions are not hoisted and can only be invoked after they have been defined.

Javascript

```
let multiplyNumbers = function(a, b) {
    return a * b;
  }
```

```
console.log(multiplyNumbers(2, 3)); // Output: 6
```

3. Arrow Functions: Arrow functions are a shorthand syntax for defining functions using the `=>` arrow. They are similar to function expressions but with a more concise syntax.

Javascript

```
let subtractNumbers = (a, b) => {

  return a - b;

}

console.log(subtractNumbers(5, 3)); // Output: 2
```

4. IIFE (Immediately Invoked Function Expressions): IIFE are self-invoking functions that are defined and immediately invoked. They are often used to create a new scope to avoid naming collisions.

```
(function() {

  let x = 5;

  let y = 3;

  console.log(x + y); // Output: 8

})();
```

5. Callback Functions: Callback functions are functions that are passed as arguments to other functions and are invoked when a certain event occurs. They are commonly used in asynchronous programming.

```
function addNumbers(a, b, callback) {

  let sum = a + b;

  callback(sum);

}

addNumbers(2, 3, function(result) {

  console.log(result); // Output: 5

});
```

6. Higher-Order Functions: Higher-order functions are functions that take one or more functions as arguments and/or return a function as a result. They are commonly used in functional programming.

```
function multiplyBy(factor) {

  return function(number) {

    return number * factor;

  }

}

let double = multiplyBy(2);

console.log(double(5)); // Output: 10
```

variable :

In JavaScript, variables are used to store data values. There are three ways to declare a variable in JavaScript:

`var`: The `var` keyword is used to declare a variable globally or locally in a function. A variable declared with `var` can be reassigned and updated.

```
var x = 5; // global scope

function myFunction() {

  var y = 10; // local scope
```

```
  console.log(x + y);
}
myFunction(); // Output: 15
```

`let`: The `let` keyword is used to declare a block-scoped variable. A variable declared with `let` can be reassigned but not redeclared.

```
let x = 5;
if (true) {
  let x = 10;
  console.log(x); // Output: 10
}
console.log(x); // Output: 5
```

`const`: The `const` keyword is used to declare a block-scoped constant. A variable declared with `const` cannot be reassigned or redeclared.

```
const PI = 3.14;
console.log(PI); // Output: 3.14
PI = 3; // Error: Assignment to constant variable.
```

JavaScript also has some reserved keywords that cannot be used as variable names. These keywords include `let`, `const`, `var`, `if`, `else`, `for`, `while`, `switch`, `case`, `default`, `break`, `continue`, `function`, `return`, `this`, `new`, `try`, `catch`, `throw`, and `finally`.

Operator

JavaScript has a variety of operators that are used to perform operations on values, including arithmetic, comparison, logical, bitwise, assignment, and others.

Here is a list of some of the operators in JavaScript:

1. Arithmetic Operators:
- `+` Addition
- `-` Subtraction
- `*` Multiplication
- `/` Division
- `%` Modulus
- `++` Increment
- `--` Decrement

2. Comparison Operators:
- `==` Equal to
- `===` Strict equal to
- `!=` Not equal to
- `!==` Strict not equal to
- `<` Less than
- `>` Greater than
- `<=` Less than or equal to
- `>=` Greater than or equal to

3. Logical Operators:
- `&&` Logical AND
- `||` Logical OR
- `!` Logical NOT

4. Bitwise Operators:
- `&` Bitwise AND
- `|` Bitwise OR

- `~` Bitwise NOT
- `^` Bitwise XOR
- `<<` Left shift
- `>>` Right shift
- `>>>` Unsigned right shift

5. Assignment Operators:

- `=` Assignment
- `+=` Addition assignment
- `-=` Subtraction assignment
- `*=` Multiplication assignment
- `/=` Division assignment
- `%=` Modulus assignment
- `<<=` Left shift assignment
- `>>=` Right shift assignment
- `&=` Bitwise AND assignment
- `^=` Bitwise XOR assignment
- `|=` Bitwise OR assignment

6. Conditional Operator:

- `? :` Ternary operator

7. Other Operators:

- `,` Comma operator
- `typeof` Type of operator
- `delete` Delete operator
- `void` Void operator
- `instanceof` Instance of operator
- `in` Property in object operator

Conditionn :

There are several types of conditions that can be used in JavaScript. Here are some of the most common ones:

1. `if` statement - Used to execute a block of code if a condition is true.

```
if (condition) {
  // code to execute if condition is true
}
```

2. `if-else` statement - Used to execute a block of code if a condition is true, and a different block of code if the condition is false.

```
if (condition) {
  // code to execute if condition is true
} else {
  // code to execute if condition is false
}
```

3. `if-else if-else` statement - Used to execute different blocks of code based on multiple conditions.

```
if (condition1) {
  // code to execute if condition1 is true
} else if (condition2) {
  // code to execute if condition2 is true
} else {
  // code to execute if all conditions are false
```

4. Ternary operator - Used to write a simple **`if-else`** statement in a single line.

```
condition ? trueValue : falseValue
```

5. Switch statement - Used to execute different blocks of code based on multiple conditions.

```
switch (value) {
  case value1:
    // code to execute if value is equal to value1
    break;
  case value2:
    // code to execute if value is equal to value2
    break;
  default:
    // code to execute if value is not equal to any of the specified values
}
```