**JAVASCRIPT NOTES**

**Numbers**

In JavaScript, numbers are represented as 64-bit floating-point values (according to the IEEE 754 standard), which means they can be integers or floating-point numbers.

Here’s how you can work with numbers in JavaScript:

**1. Defining a Number**

You can define numbers as integers or floating point (decimals):

let integer = 42; // Integer

let float = 3.14; // Floating point number

**2. Operations with Numbers**

You can perform standard mathematical operations such as addition, subtraction, multiplication, division, etc.

let sum = 5 + 3; // Addition (8)

let difference = 5 - 3; // Subtraction (2)

let product = 5 \* 3; // Multiplication (15)

let quotient = 5 / 3; // Division (1.666...)

**3. Special Numbers**

JavaScript has some special numeric values:

* **Infinity**: Represents positive infinity (result of dividing a positive number by zero).
* **-Infinity**: Represents negative infinity.
* **NaN**: Represents "Not-a-Number" (usually as a result of invalid mathematical operations).

let inf = 1 / 0; // Infinity

let negInf = -1 / 0; // -Infinity

let notANumber = 0 / 0; // NaN

**4. Parsing Numbers**

You can convert strings to numbers using parseInt() (for integers) or parseFloat() (for floating point numbers):

let num1 = parseInt("123"); // 123

let num2 = parseFloat("3.14"); // 3.14

If the conversion isn't valid, it will return NaN.

**5. Number Methods**

JavaScript provides several built-in methods to work with numbers, like Number(), toFixed(), toString(), etc.

let number = 123.456;

console.log(number.toFixed(2)); // "123.46" (rounds to two decimal places)

console.log(number.toString()); // "123.456"

console.log(Number("123.45")); // 123.45

Let me know if you want details on something specific regarding numbers in JavaScript!

**Array**

In JavaScript, a **string** is a sequence of characters used to represent text. Strings are one of the most commonly used data types in JavaScript.

### ****Creating Strings****

There are a few ways to define a string in JavaScript:

1. **Using single quotes (')**
2. **Using double quotes (")**
3. **Using template literals (backticks `\)** (Introduced in ES6)

let string1 = 'Hello, world!';

let string2 = "Hello, world!";

let string3 = `Hello, world!`; // Template literal

### ****String Operations****

#### 1. **Concatenation**

You can combine strings using the + operator.

let greeting = 'Hello';

let name = 'Alice';

let message = greeting + ', ' + name + '!';

console.log(message); // Output: "Hello, Alice!"

Alternatively, you can use **template literals** for cleaner string concatenation:

let message = `${greeting}, ${name}!`;

console.log(message); // Output: "Hello, Alice!"

#### 2. **Accessing Characters**

You can access individual characters in a string using the index. JavaScript strings are zero-indexed.

let str = 'JavaScript';

console.log(str[0]); // Output: 'J' (First character)

console.log(str[4]); // Output: 'S' (Fifth character)

You can also use the charAt() method to access a character at a given index:

console.log(str.charAt(4)); // Output: 'S'

#### 3. **String Length**

The length property returns the number of characters in a string.

let str = 'JavaScript';

console.log(str.length); // Output: 10

#### 4. **String Methods**

* **toLowerCase()** and **toUpperCase()**
  + Convert all characters to lowercase or uppercase.

let str = 'JavaScript';

console.log(str.toLowerCase()); // Output: 'javascript'

console.log(str.toUpperCase()); // Output: 'JAVASCRIPT'

* **indexOf()**
  + Returns the index of the first occurrence of a specified substring.

let str = 'JavaScript is fun';

console.log(str.indexOf('is')); // Output: 10 (index of the first occurrence of 'is')

* **slice()** or **substring()**
  + Extracts a portion of a string between two indices.

let str = 'JavaScript';

console.log(str.slice(0, 4)); // Output: 'Java' (characters from index 0 to 3)

console.log(str.substring(4, 10)); // Output: 'Script'

* **replace()**
  + Replaces a substring with a new substring.

let str = 'JavaScript is fun';

let newStr = str.replace('fun', 'awesome');

console.log(newStr); // Output: 'JavaScript is awesome'

* **split()**
  + Splits a string into an array based on a delimiter.

let str = 'JavaScript is fun';

let words = str.split(' ');

console.log(words); // Output: ['JavaScript', 'is', 'fun']

* **trim()**
  + Removes whitespace from both ends of a string.

let str = ' Hello, world! ';

console.log(str.trim()); // Output: 'Hello, world!'

### ****String Template Literals (Template Strings)****

Template literals (introduced in ES6) allow you to embed expressions within strings and create multi-line strings without needing escape characters.

let name = 'Alice';

let age = 25;

// Using template literals with embedded expressions

let greeting = `Hello, my name is ${name} and I am ${age} years old.`;

console.log(greeting); // Output: "Hello, my name is Alice and I am 25 years old."

// Multi-line string

let multiLineStr = `This is line 1

This is line 2

This is line 3`;

console.log(multiLineStr);

### ****Escaping Characters****

Sometimes, you may want to include special characters like quotes, backslashes, etc., in a string. This can be done using the escape character (\).

let str = 'He said, "Hello!"';

console.log(str); // Output: He said, "Hello!"

### ****String Immutability****

Strings in JavaScript are **immutable**, meaning once a string is created, it cannot be changed. If you modify a string, a new string is created instead.

let str = 'Hello';

str[0] = 'h'; // This will not modify the string

console.log(str); // Output: 'Hello' (not 'hello')

### ****Summary of Common String Methods****:

* .length – Returns the length of the string.
* .toLowerCase() – Converts to lowercase.
* .toUpperCase() – Converts to uppercase.
* .indexOf() – Finds the index of a substring.
* .slice() or .substring() – Extracts part of a string.
* .replace() – Replaces part of a string.
* .split() – Splits a string into an array.
* .trim() – Removes whitespace from both ends.
* .charAt() – Returns the character at a given index.
* .includes() – Checks if a string contains a specified substring.

Let me know if you need help with something specific regarding strings!

**Object**

In JavaScript, an **object** is a collection of properties, where each property is a key-value pair. The **key** (also called a **property name**) is usually a string, and the **value** can be any valid JavaScript data type, including other objects, arrays, or even functions.

**Creating an Object**

There are two primary ways to create an object in JavaScript:

1. **Object Literal Notation** (most common):
2. let person = {
3. name: 'Alice',
4. age: 25,
5. isStudent: false
6. };
7. **Using the new Object() Syntax**:
8. let person = new Object();
9. person.name = 'Alice';
10. person.age = 25;
11. person.isStudent = false;

**Accessing Object Properties**

You can access the properties of an object using either dot notation or bracket notation:

1. **Dot Notation**:
2. console.log(person.name); // Output: 'Alice'
3. console.log(person.age); // Output: 25
4. **Bracket Notation** (useful when property names have spaces or are dynamic):
5. console.log(person['name']); // Output: 'Alice'
6. console.log(person['age']); // Output: 25

**Modifying Object Properties**

You can modify an object’s properties by simply assigning a new value to a property:

person.age = 26; // Modify the 'age' property

console.log(person.age); // Output: 26

Or using **bracket notation**:

person['name'] = 'Bob'; // Modify the 'name' property

console.log(person.name); // Output: 'Bob'

**Adding New Properties**

You can add new properties to an object using dot notation or bracket notation:

person.gender = 'female'; // Add a new property

console.log(person.gender); // Output: 'female'

person['country'] = 'USA'; // Another way to add a property

console.log(person.country); // Output: 'USA'

**Deleting Properties**

To delete a property from an object, you can use the delete keyword:

delete person.isStudent; // Remove the 'isStudent' property

console.log(person.isStudent); // Output: undefined

**Nested Objects**

Objects can also contain other objects as values:

let person = {

name: 'Alice',

address: {

street: '123 Main St',

city: 'New York',

zipCode: 10001

}

};

console.log(person.address.city); // Output: 'New York'

**Accessing Object Methods (Functions as Values)**

Objects can store functions as values. These are often called **methods**:

let person = {

name: 'Alice',

greet: function() {

console.log('Hello, ' + this.name);

}

};

person.greet(); // Output: 'Hello, Alice'

You can also use **arrow functions** for methods:

let person = {

name: 'Alice',

greet: () => {

console.log('Hello, ' + this.name); // Note: 'this' doesn't work as expected in arrow functions

}

};

person.greet(); // Output: 'Hello, undefined' (since 'this' is not bound as expected)

For methods, it's recommended to use traditional function expressions to ensure this works correctly.

**Checking if a Property Exists**

You can check if an object has a certain property using the in operator or hasOwnProperty() method:

1. **Using the in operator:**
2. console.log('name' in person); // Output: true
3. console.log('email' in person); // Output: false
4. **Using the hasOwnProperty() method:**
5. console.log(person.hasOwnProperty('name')); // Output: true
6. console.log(person.hasOwnProperty('email')); // Output: false

**Iterating Over Object Properties**

You can loop through the properties of an object using for...in or Object.keys():

1. **Using for...in loop:**
2. for (let key in person) {
3. console.log(key + ': ' + person[key]);
4. }
5. // Output:
6. // name: Alice
7. // age: 26
8. // gender: female
9. // country: USA
10. **Using Object.keys() (Returns an array of object keys):**
11. Object.keys(person).forEach(key => {
12. console.log(key + ': ' + person[key]);
13. });
14. // Output:
15. // name: Alice
16. // age: 26
17. // gender: female
18. // country: USA

**Object Methods**

JavaScript provides several built-in methods for working with objects:

1. **Object.keys()** – Returns an array of an object's own enumerable property names.
2. let keys = Object.keys(person);
3. console.log(keys); // Output: ['name', 'age', 'gender', 'country']
4. **Object.values()** – Returns an array of an object's values.
5. let values = Object.values(person);
6. console.log(values); // Output: ['Alice', 26, 'female', 'USA']
7. **Object.entries()** – Returns an array of key-value pairs.
8. let entries = Object.entries(person);
9. console.log(entries);
10. // Output: [['name', 'Alice'], ['age', 26], ['gender', 'female'], ['country', 'USA']]
11. **Object.assign()** – Copies all properties from one or more source objects to a target object.
12. let person2 = { name: 'Bob', age: 30 };
13. let personCopy = Object.assign({}, person, person2);
14. console.log(personCopy);
15. // Output: { name: 'Bob', age: 30, gender: 'female', country: 'USA' }

**Summary:**

* **Objects** are used to store collections of data in the form of key-value pairs.
* You can **create** objects using object literal notation {} or new Object().
* **Properties** of an object are accessed using dot notation (object.property) or bracket notation (object['property']).
* **Methods** are functions stored as properties within an object.
* Objects are **mutable**, meaning you can modify, add, or delete properties after the object is created.

Let me know if you'd like further examples or explanations on objects!

**Conditionals**

In JavaScript, **conditionals** are used to perform different actions based on different conditions. The most common conditional statements are:

1. **if statement**
2. **if...else statement**
3. **else if statement**
4. **switch statement**

**1. if Statement**

The if statement is used to execute a block of code if a specified condition is true.

let age = 18;

if (age >= 18) {

console.log("You are an adult.");

}

In this example, since age is 18, the condition evaluates to true, and the message "You are an adult." is printed.

**2. if...else Statement**

The if...else statement allows you to specify what to do if the condition is true and what to do if it's false.

let age = 16;

if (age >= 18) {

console.log("You are an adult.");

} else {

console.log("You are a minor.");

}

Here, since age is 16, the condition is false, so the message "You are a minor." will be printed.

**3. else if Statement**

The else if statement allows you to test multiple conditions. You can chain multiple else if conditions after an if block.

let age = 20;

if (age < 13) {

console.log("You are a child.");

} else if (age >= 13 && age < 18) {

console.log("You are a teenager.");

} else if (age >= 18 && age < 60) {

console.log("You are an adult.");

} else {

console.log("You are a senior.");

}

In this case, the condition age >= 18 && age < 60 evaluates to true, so the message "You are an adult." is printed.

**4. switch Statement**

The switch statement is another way to handle multiple conditions, often used when there are many possible values for a single variable.

let day = 3;

let dayName;

switch (day) {

case 1:

dayName = "Monday";

break;

case 2:

dayName = "Tuesday";

break;

case 3:

dayName = "Wednesday";

break;

case 4:

dayName = "Thursday";

break;

case 5:

dayName = "Friday";

break;

case 6:

dayName = "Saturday";

break;

case 7:

dayName = "Sunday";

break;

default:

dayName = "Invalid day";

}

console.log(dayName); // Output: 'Wednesday'

In this example:

* The switch statement checks the value of the day variable.
* When it matches 3, the corresponding case block is executed, and dayName is set to "Wednesday".
* The break keyword is used to exit the switch statement once a match is found.

The **default** case is executed if no match is found, like an "else" in an if block.

**Comparison Operators in Conditionals**

JavaScript uses several comparison operators to evaluate conditions:

* **==**: Equal to (checks value)
* **===**: Strict equal to (checks value and type)
* **!=**: Not equal to (checks value)
* **!==**: Strict not equal to (checks value and type)
* **>**: Greater than
* **<**: Less than
* **>=**: Greater than or equal to
* **<=**: Less than or equal to

**Logical Operators**

Logical operators are used to combine multiple conditions.

* **&&**: AND (both conditions must be true)
* **||**: OR (at least one condition must be true)
* **!**: NOT (reverses the condition)

Example using **logical operators**:

let age = 25;

let hasLicense = true;

if (age >= 18 && hasLicense) {

console.log("You are allowed to drive.");

} else {

console.log("You are not allowed to drive.");

}

In this case, the condition checks if the person is 18 or older **and** has a driving license. If both conditions are true, "You are allowed to drive." will be printed.

**Ternary Operator (Shorthand for if...else)**

The ternary operator is a shorthand for the if...else statement. It is often used when you need to assign a value based on a condition.

let age = 20;

let message = (age >= 18) ? "You are an adult." : "You are a minor.";

console.log(message); // Output: "You are an adult."

The syntax is:

condition ? expressionIfTrue : expressionIfFalse;

**Short-Circuiting with Logical Operators**

In JavaScript, you can use **short-circuiting** with the logical operators && and ||:

* **&& (AND)**: If the first condition is false, the second condition is not evaluated.
* **|| (OR)**: If the first condition is true, the second condition is not evaluated.

Example:

let name = "";

let defaultName = "Guest";

let displayName = name || defaultName;

console.log(displayName); // Output: 'Guest'

In this example, since name is an empty string (which is falsy), the value of defaultName ("Guest") is used.

**Summary of Key Concepts:**

* **if statement**: Executes code if a condition is true.
* **if...else**: Executes one block of code if the condition is true and another block if it's false.
* **else if**: Allows you to test multiple conditions.
* **switch**: A way to handle multiple possible conditions based on the value of a single expression.
* **Comparison operators**: Used to compare values (==, ===, !=, >, <, etc.).
* **Logical operators**: Used to combine multiple conditions (&&, ||, !).
* **Ternary operator**: A shorthand for if...else statements.
* **Short-circuiting**: Logical operators can help avoid unnecessary evaluation of expressions.

Let me know if you need more examples or explanations!

**Loops**

In JavaScript, **loops** are used to execute a block of code repeatedly based on a specified condition. There are several types of loops in JavaScript, each with its own specific use case:

### ****1.**** for ****Loop****

The for loop is one of the most commonly used loops. It is generally used when you know in advance how many times you want to loop through a block of code.

#### Syntax:

for (initialization; condition; increment) {

// Code to be executed

}

* **Initialization**: Sets up the loop variable (e.g., let i = 0).
* **Condition**: The loop will run as long as this condition is true.
* **Increment**: This defines how the loop variable should be updated after each iteration.

#### Example:

for (let i = 0; i < 5; i++) {

console.log(i); // Outputs: 0, 1, 2, 3, 4

}

In this example:

* i = 0: Initialization.
* i < 5: The loop runs as long as i is less than 5.
* i++: After each iteration, i is incremented by 1.

### ****2.**** while ****Loop****

The while loop is used when you want to execute a block of code as long as a condition is true. If the condition is initially false, the code inside the loop won't be executed at all.

#### Syntax:

while (condition) {

// Code to be executed

}

#### Example:

let i = 0;

while (i < 5) {

console.log(i); // Outputs: 0, 1, 2, 3, 4

i++; // Increment i by 1

}

In this example, the loop will run as long as i is less than 5, and i is incremented by 1 after each iteration.

### ****3.**** do...while ****Loop****

The do...while loop is similar to the while loop, but the difference is that the code inside the loop is executed **at least once** before the condition is tested, even if the condition is initially false.

#### Syntax:

do {

// Code to be executed

} while (condition);

#### Example:

let i = 0;

do {

console.log(i); // Outputs: 0, 1, 2, 3, 4

i++; // Increment i by 1

} while (i < 5);

In this example, the loop runs even if i is initially 0 because the code inside the loop is executed before the condition is tested.

### ****4.**** for...in ****Loop****

The for...in loop is used to loop through the **properties** of an object or elements of an array (although it’s generally used for objects).

#### Syntax:

for (let key in object) {

// Code to be executed

}

#### Example (Looping through an object):

let person = {

name: 'Alice',

age: 25,

country: 'USA'

};

for (let key in person) {

console.log(key + ': ' + person[key]);

}

// Outputs:

// name: Alice

// age: 25

// country: USA

In this example, key represents each property name of the person object, and we access the value using person[key].

#### Example (Looping through an array):

let fruits = ['Apple', 'Banana', 'Cherry'];

for (let index in fruits) {

console.log(fruits[index]);

}

// Outputs:

// Apple

// Banana

// Cherry

### ****5.**** for...of ****Loop****

The for...of loop is used to iterate over the **values** of an iterable (e.g., an array, string, etc.). It’s often more convenient for iterating through arrays or other iterable data structures.

#### Syntax:

for (let value of iterable) {

// Code to be executed

}

#### Example (Looping through an array):

let fruits = ['Apple', 'Banana', 'Cherry'];

for (let fruit of fruits) {

console.log(fruit);

}

// Outputs:

// Apple

// Banana

// Cherry

In this example, fruit represents each value in the fruits array.

#### Example (Looping through a string):

let name = "Alice";

for (let char of name) {

console.log(char);

}

// Outputs:

// A

// l

// i

// c

// e

### ****6.**** break ****Statement****

The break statement is used to exit the current loop, even if the loop's condition has not been satisfied.

#### Example:

for (let i = 0; i < 10; i++) {

if (i === 5) {

break; // Exit the loop when i is 5

}

console.log(i); // Outputs: 0, 1, 2, 3, 4

}

### ****7.**** continue ****Statement****

The continue statement is used to skip the current iteration of a loop and continue with the next iteration.

#### Example:

for (let i = 0; i < 5; i++) {

if (i === 3) {

continue; // Skip when i is 3

}

console.log(i); // Outputs: 0, 1, 2, 4

}

In this case, the continue statement causes the loop to skip the iteration where i is 3.

### ****Summary of Loop Types:****

1. **for loop**: Best used when you know the number of iterations beforehand.
2. **while loop**: Best used when the number of iterations is unknown and depends on a condition.
3. **do...while loop**: Similar to while, but ensures the loop runs at least once.
4. **for...in loop**: Loops through the properties of an object (or array indices).
5. **for...of loop**: Loops through the values of an iterable (like arrays or strings).
6. **break**: Terminates the loop.
7. **continue**: Skips the current iteration and moves to the next one.

Each type of loop has its own use case, depending on what you're trying to achieve. Let me know if you need more examples or explanations!

**Functions**

In JavaScript, **functions** are blocks of reusable code that can be executed when called. Functions help to organize your code, make it modular, and avoid repetitive code. Functions can accept inputs (parameters), perform some logic, and return an output.

### ****1. Function Declaration (Function Statement)****

A **function declaration** is the most common way to define a function in JavaScript. It uses the function keyword followed by the function name, a list of parameters (optional), and the function body.

#### Syntax:

function functionName(parameters) {

// Code to be executed

return result; // Optional return value

}

#### Example:

function greet(name) {

console.log("Hello, " + name);

}

greet("Alice"); // Output: 'Hello, Alice'

In this example, the function greet takes a single parameter name, and prints a greeting message. The function is called with "Alice", and the message is printed.

### ****2. Function Expression****

A **function expression** defines a function inside an expression (typically assigned to a variable). These functions are anonymous (do not have a name) unless explicitly given one.

#### Syntax:

let functionName = function(parameters) {

// Code to be executed

return result;

};

#### Example:

let greet = function(name) {

console.log("Hello, " + name);

};

greet("Bob"); // Output: 'Hello, Bob'

In this example, the function is assigned to the variable greet, and then it is called using the variable name.

### ****3. Arrow Functions****

Arrow functions are a shorter syntax for defining functions. They also have some differences in how they handle the this keyword, but for simple cases, they provide a cleaner, more concise way to define functions.

#### Syntax:

let functionName = (parameters) => {

// Code to be executed

return result;

};

If the function has only one expression, you can omit the curly braces and return keyword.

#### Example:

let greet = (name) => {

console.log("Hello, " + name);

};

greet("Charlie"); // Output: 'Hello, Charlie'

For functions with a single expression:

let greet = name => console.log("Hello, " + name);

greet("David"); // Output: 'Hello, David'

### ****4. Function Parameters and Arguments****

Functions can accept parameters (inputs) when called. These parameters can be used within the function to perform specific tasks. JavaScript functions can also handle default values for parameters.

#### Example with parameters:

function sum(a, b) {

return a + b;

}

console.log(sum(3, 5)); // Output: 8

#### Default Parameters:

You can set default values for parameters in case the caller does not pass a value.

function greet(name = "Guest") {

console.log("Hello, " + name);

}

greet("Alice"); // Output: 'Hello, Alice'

greet(); // Output: 'Hello, Guest'

### ****5. Return Statement****

The return statement is used to return a value from the function. If no return value is specified, the function will return undefined by default.

#### Example:

function multiply(x, y) {

return x \* y;

}

let result = multiply(4, 5);

console.log(result); // Output: 20

### ****6. Function Scope****

Each function in JavaScript has its own **local scope**, meaning variables declared inside a function are not accessible outside of that function.

#### Example of function scope:

function test() {

let a = 10; // Local variable

console.log(a); // Output: 10

}

test();

console.log(a); // Error: a is not defined

In this example, a is defined inside the test function and cannot be accessed outside of it.

### ****7. Function Hoisting****

In JavaScript, **function declarations** are hoisted, meaning they are moved to the top of the code during the compilation phase. This allows you to call functions before they are defined in the code.

#### Example of function hoisting:

greet("Alice"); // Output: 'Hello, Alice'

function greet(name) {

console.log("Hello, " + name);

}

However, **function expressions** are **not hoisted**. If you try to call a function expression before it’s defined, you will get an error.

greet("Alice"); // Error: greet is not a function

let greet = function(name) {

console.log("Hello, " + name);

};

### ****8. Function as Arguments (Callback Functions)****

Functions in JavaScript can be passed as arguments to other functions. This is a powerful feature that allows for more flexible and dynamic code.

#### Example of a function as an argument:

function greet(name, callback) {

console.log("Hello, " + name);

callback();

}

function sayGoodbye() {

console.log("Goodbye!");

}

greet("Alice", sayGoodbye);

// Output:

// Hello, Alice

// Goodbye!

In this example, sayGoodbye is passed as a callback function to greet.

### ****9. Anonymous Functions****

Anonymous functions are functions that don’t have a name. They are often used in situations where a function is passed as an argument or used temporarily.

#### Example:

setTimeout(function() {

console.log("This is an anonymous function.");

}, 1000);

In this example, an anonymous function is passed to setTimeout and executed after 1 second.

### ****10. Closures****

A **closure** is a function that "remembers" its lexical scope, even when the function is executed outside that scope. Closures allow you to create private variables and functions.

#### Example of closure:

function outer() {

let count = 0;

return function inner() {

count++;

console.log(count);

};

}

let counter = outer();

counter(); // Output: 1

counter(); // Output: 2

counter(); // Output: 3

In this example, the inner function forms a closure with the count variable in the outer function’s scope.

### ****Summary of Key Concepts:****

* **Function Declaration**: A function defined with the function keyword.
* **Function Expression**: A function assigned to a variable (typically anonymous).
* **Arrow Function**: A more concise way to define functions with a different handling of this.
* **Parameters and Arguments**: Functions accept parameters (inputs) and can return values.
* **Return Statement**: Used to return a value from the function.
* **Scope**: Variables declared inside a function have local scope.
* **Hoisting**: Function declarations are hoisted, but function expressions are not.
* **Callback Functions**: Functions can be passed as arguments to other functions.
* **Anonymous Functions**: Functions without names, often used for short-term or one-time usage.
* **Closures**: Functions that maintain access to their lexical scope.

Let me know if you'd like more examples or further explanation on any part of functions!