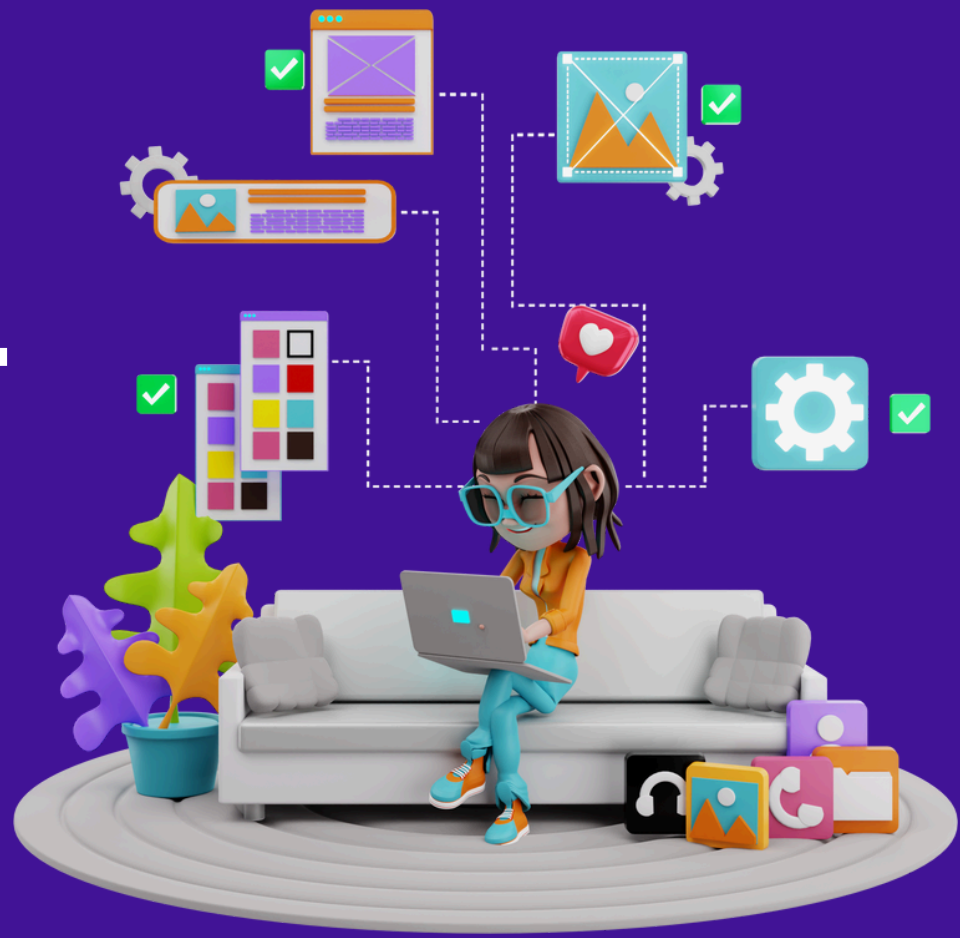
WEB DEVELOPMENT COURSE

LEARN FROM SCRATCH TO ADVANCED

JAVA SCRIPT



INTRODUCTION

JS INTRODUCTION

TODAY, I AM CREATING THIS JAVASCRIPT TUTORIAL TO PROVIDE YOU WITH A RESOURCE THAT WILL HELP YOU LEARN JAVASCRIPT. ALSO, YOU CAN USE THIS TUTORIAL AS A REFERENCE LATER TO LOOKUP VARIOUS CONCEPTS AND CODE.

MY NAME IS HARRY AND I AM A COMPUTER PROGRAMMER AND A YOUTUBER FROM CODEWITHHARRY YOUTUBE CHANNEL. I WANT TO START THIS TUTORIAL BY WHY WE NEED JAVASCRIPT AT THE FIRST PLACE.

IMAGINE A WEBSITE AS A HOUSE. HTML IS LIKE THE BRICKS AND WALLS THAT GIVE IT STRUCTURE. CSS IS LIKE THE PAINT AND DECORATIONS THAT MAKE IT LOOK NICE. BUT WITHOUT JAVASCRIPT, THE HOUSE WON'T HAVE ANY LIGHTS OR RUNNING WATER—NOTHING WOULD WORK OR MOVE. JAVASCRIPT IS WHAT MAKES A WEBSITE INTERACTIVE. IT'S LIKE ADDING ELECTRICITY TO THE HOUSE. IT LETS YOU CLICK BUTTONS TO OPEN DOORS, TURN ON LIGHTS, OR EVEN PLAY MUSIC. SO, WITHOUT JAVASCRIPT, A WEBSITE WOULD BE LIKE A HOUSE WHERE NOTHING REALLY HAPPENS—YOU COULD LOOK AT IT. THE VIDEO BELOW SHOWS A MODAL WHICH IS CREATED USING JAVASCRIPT

Live demo

Toggle a working modal demo by clicking the button below. It will slide down and fade in from the top of the page.

Launch demo modal

```
<!-- Button trigger modal -->
<button type="button" class="btn btn-primary" data-bs-toggle="modal" data-bs-target="#exampleMo
  Launch demo modal
</button>

<!-- Modal -->
<div class="modal fade" id="exampleModal" tabindex="-1" aria-labelledby="exampleModalLabel" ari
  <div class="modal-dialog">
    <div class="modal-content">
      <div class="modal-header">
        <h1 class="modal-title fs-5" id="exampleModalLabel">Modal title</h1>
        <button type="button" class="btn-close" data-bs-dismiss="modal" aria-label="Close"></bu
      </div>
      <div class="modal-body">
        ...
      </div>
      <div class="modal-footer">
        <button type="button" class="btn btn-secondary" data-bs-dismiss="modal">Close</button>
        <button type="button" class="btn btn-primary">Save changes</button>
      </div>
    </div>
  </div>
</div>
```

Static backdrop

When backdrop is set to static, the modal will not close when clicking outside of it. Click the button below to try it.

JAVASCRIPT IS USED FOR FRONTEND PROGRAMMING BUT IT IS ALSO USED FOR BACKEND PROGRAMMING USING NODE.JS. LETS TAKE A STEP BACK AND UNDERSTAND ALL THESE CONCEPTS ONE BY ONE

WHAT IS PROGRAMMING?

IT IS A WAY TO TALK TO COMPUTERS. A LANGUAGE LIKE HINDI, ENGLISH, OR BENGALI CAN BE USED TO TALK TO A HUMAN BUT FOR COMPUTERS, WE NEED STRAIGHTFORWARD INSTRUCTIONS.
PROGRAMMING IS THE ACT OF CONSTRUCTING A PROGRAM, A SET OF PRECISE INSTRUCTIONS TELLING A COMPUTER WHAT TO DO.
PROGRAMMING IS THE PROCESS OF DESIGNING, WRITING, TESTING, DEBUGGING, AND MAINTAINING THE SOURCE CODE OF COMPUTER PROGRAMS. IT INVOLVES THE USE OF PROGRAMMING LANGUAGES AND VARIOUS TOOLS TO CREATE SOFTWARE PROGRAMS THAT CAN BE RUN ON A COMPUTER OR OTHER DEVICES.

WHAT IS ECMASCRIPT?

ECMASCRIPT IS A STANDARD ON WHICH JAVASCRIPT IS BASED!
IT WAS CREATED TO ENSURE THAT DIFFERENT DOCUMENTS ON JAVASCRIPT ARE TALKING ABOUT THE SAME LANGUAGE.

ECMASCRIPT VERSIONS:

BEGINNING IN 1997, JAVASCRIPT HAS EVOLVED INTO MANY VERSIONS.

- ECMASCRIPT1 OR ES1 WAS FIRST RELEASED IN 1997.
- ECMASCRIPT 2015 ALSO KNOWN AS ES6 WAS RELEASED IN 2015 AND A MAJOR REVISION TO JAVASCRIPT WAS MADE.
- THE LATEST VERSION OF JAVASCRIPT WILL BE ECMASCRIPT2022 (ES13) IN 2022.

WHAT IS JAVASCRIPT?

- JAVASCRIPT IS A LIGHTWEIGHT, OOP LANGUAGE.
- IT IS A SCRIPTING LANGUAGE FOR WEB PAGES.
- IT IS USED TO ADD INTERACTIVITY AND DYNAMIC EFFECTS TO WEB PAGES.
- ".JS" IS THE EXTENSION.
- NOWADAYS USED IN SERVER-SIDE DEVELOPMENT.
- JS FRONTEND FRAMEWORKS: REACT, ANGULAR, VUE.
- JS BACKEND FRAMEWORKS: EXPRESS, NODE.

IN THIS TUTORIAL, WE WILL LEARN JAVASCRIPT IN DEPTH

JS EXECUTION

DO WE NEED TO INSTALL JAVASCRIPT?

THE ANSWER IS **NO**.
JAVASCRIPT IS PRESENT EVERYWHERE IN BROWSERS, ON YOUR PHONE, ETC.

HOW TO EXECUTE JAVASCRIPT?

JAVASCRIPT CAN BE EXECUTED RIGHT INSIDE ONE'S BROWSER. YOU CAN OPEN THE JS CONSOLE AND START WRITING JS THERE.
ANOTHER WAY TO EXECUTE JS IS A RUNTIME LIKE NODE.JS WHICH CAN BE INSTALLED AND USED TO RUN JAVASCRIPT CODE.
YET ANOTHER WAY TO EXECUTE JAVASCRIPT IS BY INSERTING IT INSIDE THE <SCRIPT> TAG OF AN HTML DOCUMENT.

WHAT IS JAVASCRIPT?

JAVASCRIPT IS A PROGRAMMING LANGUAGE THAT IS COMMONLY USED IN WEB DEVELOPMENT. IT IS A CLIENT-SIDE LANGUAGE, WHICH MEANS THAT IT IS EXECUTED BY THE USER'S WEB BROWSER RATHER THAN THE WEB SERVER. THIS ALLOWS JAVASCRIPT TO INTERACT WITH THE USER AND CREATE DYNAMIC, INTERACTIVE WEB PAGES.
JAVASCRIPT IS OFTEN USED IN COMBINATION WITH HTML AND CSS TO CREATE WEB PAGES THAT ARE MORE INTERACTIVE AND ENGAGING.

GETTING STARTED WITH JAVASCRIPT

TO START USING JAVASCRIPT, YOU'LL NEED A TEXT EDITOR AND A WEB BROWSER. THERE ARE MANY TEXT EDITORS AVAILABLE, SUCH AS SUBLIME TEXT, ATOM, AND VISUAL STUDIO CODE, WHICH ARE ALL POPULAR CHOICES AMONG DEVELOPERS.

ONCE YOU HAVE A TEXT EDITOR SET UP, YOU CAN START WRITING JAVASCRIPT CODE. TO DO THIS, YOU'LL NEED TO CREATE A NEW FILE WITH A .JS EXTENSION AND THEN TYPE YOUR CODE INTO THE FILE. YOU CAN THEN SAVE THE FILE AND OPEN IT IN YOUR WEB BROWSER TO SEE THE RESULTS.

NODE.JS INSTALLATION

NODE.JS INSTALLATION

NODE.JS IS AN OPEN-SOURCE, CROSS-PLATFORM JAVASCRIPT RUNTIME ENVIRONMENT THAT EXECUTES JAVASCRIPT CODE OUTSIDE OF A WEB BROWSER. IT IS COMMONLY USED FOR BUILDING SERVER-SIDE APPLICATIONS, COMMAND-LINE TOOLS, AND OTHER TYPES OF SCALABLE NETWORK PROGRAMS.

TO INSTALL NODE.JS, YOU CAN FOLLOW THESE STEPS:

1. DOWNLOAD THE INSTALLER: GO TO THE OFFICIAL NODE.JS WEBSITE (HTTPS://NODEJS.ORG/) AND CLICK THE "DOWNLOAD" BUTTON TO DOWNLOAD THE LATEST VERSION OF THE NODE.JS INSTALLER.

2.RUN THE INSTALLER: DOUBLE-CLICK THE DOWNLOADED INSTALLER FILE TO START THE INSTALLATION PROCESS. FOLLOW THE PROMPTS TO COMPLETE THE INSTALLATION.
3.VERIFY THE INSTALLATION: TO VERIFY THAT NODE.JS HAS BEEN INSTALLED SUCCESSFULLY, OPEN A TERMINAL OR COMMAND PROMPT AND TYPE THE FOLLOWING COMMAND:

RUN THE FOLLOWING COMMAND:

```
node -v
```

THIS SHOULD OUTPUT THE VERSION NUMBER OF NODE.JS THAT YOU HAVE INSTALLED.
THAT'S IT! YOU SHOULD NOW HAVE NODE.JS INSTALLED ON YOUR SYSTEM AND BE ABLE TO RUN JAVASCRIPT CODE USING THE "NODE" COMMAND.
I HOPE THIS HELPS. LET ME KNOW IF YOU HAVE ANY QUESTIONS.

SCRIPT USING NODE.JS

TO RUN JAVASCRIPT WITH VISUAL STUDIO CODE (VS CODE), YOU WILL NEED TO FOLLOW THESE STEPS:

1. INSTALL VS CODE: IF YOU DON'T ALREADY HAVE IT, YOU CAN DOWNLOAD AND INSTALL VS CODE FROM THE OFFICIAL WEBSITE (HTTPS://CODE.VISUALSTUDIO.COM/).
2. CREATE A NEW JAVASCRIPT FILE: OPEN VS CODE AND CREATE A NEW FILE WITH A .JS EXTENSION. YOU CAN DO THIS BY GOING TO FILE > NEW FILE OR BY USING THE SHORTCUT CTRL + N.
3. WRITE YOUR JAVASCRIPT CODE: TYPE YOUR JAVASCRIPT CODE INTO THE FILE AND SAVE IT.

4. OPEN THE COMMAND PALETTE: YOU CAN OPEN THE COMMAND PALETTE BY PRESSING CTRL + SHIFT + P OR BY GOING TO VIEW > COMMAND PALETTE.
5.RUN THE JAVASCRIPT FILE: IN THE COMMAND PALETTE, TYPE "RUN JAVASCRIPT" AND SELECT "RUN JAVASCRIPT FILE IN THE TERMINAL" FROM THE LIST OF OPTIONS. THIS WILL OPEN A TERMINAL WINDOW AND RUN YOUR JAVASCRIPT FILE.
6.VIEW THE OUTPUT: THE OUTPUT OF YOUR JAVASCRIPT CODE WILL BE DISPLAYED IN THE TERMINAL WINDOW.

ALTERNATIVELY, YOU CAN ALSO RUN JAVASCRIPT CODE DIRECTLY IN THE TERMINAL BY USING A COMMAND-LINE INTERPRETER SUCH AS NODE.JS. TO DO THIS, YOU WILL NEED TO INSTALL NODE.JS AND THEN RUN YOUR JAVASCRIPT FILE USING THE "NODE" COMMAND, FOLLOWED BY THE NAME OF THE FILE. FOR EXAMPLE:

```
node myfile.js
```

JAVASCRIPT VARIABLES

WHAT ARE VARIABLES?

IN JAVASCRIPT, VARIABLES ARE USED TO STORE DATA. THEY ARE AN ESSENTIAL PART OF ANY PROGRAMMING LANGUAGE, AS THEY ALLOW YOU TO STORE, RETRIEVE, AND MANIPULATE DATA IN YOUR PROGRAMS.

THERE ARE A FEW DIFFERENT WAYS TO DECLARE VARIABLES IN JAVASCRIPT, EACH WITH ITS OWN SYNTAX AND RULES. IN THIS BLOG POST, WE'LL TAKE A LOOK AT THE DIFFERENT TYPES OF VARIABLES AVAILABLE IN JAVASCRIPT AND HOW TO USE THEM.

DECLARING VARIABLES

TO DECLARE A VARIABLE IN JAVASCRIPT, YOU USE THE "VAR" KEYWORD FOLLOWED BY THE NAME OF THE VARIABLE. FOR EXAMPLE:

```
var x;
```

TO DECLARE A VARIABLE IN JAVASCRIPT, YOU USE THE "VAR" KEYWORD FOLLOWED BY THE NAME OF THE VARIABLE. FOR EXAMPLE:

```
var x = 10;
```

IN JAVASCRIPT, YOU CAN ALSO USE THE "LET" AND "CONST" KEYWORDS TO DECLARE VARIABLES. THE "LET" KEYWORD IS USED TO DECLARE A VARIABLE THAT CAN BE REASSIGNED LATER, WHILE THE "CONST" KEYWORD IS USED TO DECLARE A VARIABLE THAT CANNOT BE REASSIGNED. FOR EXAMPLE:

```
let y = 20;
const z = 30;
```

IN THIS EXAMPLE, "Y" IS A VARIABLE THAT CAN BE REASSIGNED, WHILE "Z" IS A CONSTANT THAT CANNOT BE REASSIGNED.

DATA TYPES

IN JAVASCRIPT, THERE ARE SEVERAL DATA TYPES THAT YOU CAN USE TO STORE DIFFERENT TYPES OF DATA. SOME COMMON DATA TYPES INCLUDE:

- NUMBERS (E.G. 10, 3.14)
- STRINGS (E.G. "HELLO", 'WORLD')
- BOOLEANS (E.G. TRUE, FALSE)
- ARRAYS (E.G. [1, 2, 3])
- OBJECTS (E.G. { NAME: "JOHN", AGE: 30 })

VARIABLE NAMING RULES

JAVASCRIPT IS A DYNAMICALLY-TYPED LANGUAGE, WHICH MEANS THAT YOU DON'T HAVE TO SPECIFY THE DATA TYPE OF A VARIABLE WHEN YOU DECLARE IT. THE DATA TYPE OF A VARIABLE IS DETERMINED BY THE VALUE THAT IS ASSIGNED TO IT. FOR EXAMPLE:

```
var x = 10; // x is a number
var y = "hello"; // y is a string
var z = [1, 2, 3]; // z is an array
```

VARIABLE NAMING RULES

THERE ARE A FEW RULES THAT YOU NEED TO FOLLOW WHEN NAMING VARIABLES IN JAVASCRIPT:

- VARIABLE NAMES CAN ONLY CONTAIN LETTERS, DIGITS, UNDERSCORES, AND DOLLAR SIGNS.
- VARIABLE NAMES CANNOT START WITH A DIGIT.
- VARIABLE NAMES ARE CASE-SENSITIVE.

THERE ARE A FEW RULES THAT YOU NEED TO FOLLOW WHEN NAMING VARIABLES IN JAVASCRIPT:

USING VARIABLES

ONCE YOU HAVE DECLARED A VARIABLE, YOU CAN USE IT TO STORE AND RETRIEVE DATA IN YOUR PROGRAM. FOR EXAMPLE:

```
var x = 10;
console.log(x); // prints 10
x = "hello";
console.log(x); // prints "hello"
```

Copy

YOU CAN ALSO PERFORM VARIOUS OPERATIONS ON VARIABLES, SUCH AS MATHEMATICAL CALCULATIONS, STRING CONCATENATION, AND MORE. FOR EXAMPLE:

```
var x = 10;
var y = 20;
var z = x + y; // z is 30var str1 = "hello";var str2 = "world";var str3 = str1 + " " + str2; // str3 is "hello world"
```

PRIMITIVES AND OBJECTS

IN JAVASCRIPT, THERE ARE TWO MAIN TYPES OF DATA: PRIMITIVES AND OBJECTS.

PRIMITIVES

PRIMITIVES ARE THE SIMPLEST AND MOST BASIC DATA TYPES IN JAVASCRIPT. THEY INCLUDE:

- NUMBERS (E.G. 10, 3.14)
- STRINGS (E.G. "HELLO", 'WORLD')
- BOOLEANS (E.G. TRUE, FALSE)
- NULL (A SPECIAL VALUE THAT REPRESENTS AN ABSENCE OF VALUE)
- UNDEFINED (A SPECIAL VALUE THAT REPRESENTS AN UNINITIALIZED VARIABLE)

PRIMITIVES ARE IMMUTABLE, WHICH MEANS THAT ONCE THEY ARE CREATED, THEY CANNOT BE CHANGED. FOR EXAMPLE:

```
let x = 10;
x = 20; // x is now 20
```

IN THIS EXAMPLE, THE VALUE OF "X" IS CHANGED FROM 10 TO 20. HOWEVER, THIS DOES NOT CHANGE THE VALUE OF THE PRIMITIVE ITSELF, BUT RATHER CREATES A NEW PRIMITIVE WITH THE VALUE OF 20.

OBJECTS

OBJECTS ARE MORE COMPLEX DATA TYPES IN JAVASCRIPT AND ARE USED TO REPRESENT REAL-WORLD OBJECTS OR ABSTRACT CONCEPTS. THEY ARE COMPOSED OF KEY-VALUE PAIRS, WHERE THE KEYS ARE STRINGS AND THE VALUES CAN BE ANY DATA TYPE (INCLUDING PRIMITIVES AND OTHER OBJECTS).

OBJECTS ARE MUTABLE, WHICH MEANS THAT THEY CAN BE CHANGED AFTER THEY ARE CREATED.
FOR EXAMPLE:

```
let obj = { name: "John", age: 30 };
obj.age = 31; // the age property of obj is now 31
```

IN THIS EXAMPLE, THE "AGE" PROPERTY OF THE "OBJ" OBJECT IS CHANGED FROM 30 TO 31. THIS CHANGES THE VALUE OF THE OBJECT ITSELF, RATHER THAN CREATING A NEW OBJECT.
THERE ARE SEVERAL OTHER DATA TYPES IN JAVASCRIPT THAT ARE CLASSIFIED AS OBJECTS, INCLUDING ARRAYS, FUNCTIONS, AND DATES. THESE DATA TYPES BEHAVE SIMILARLY TO OBJECTS IN THAT THEY ARE MUTABLE AND CAN BE MODIFIED AFTER THEY ARE CREATED.

CONCLUSION

IN SUMMARY, PRIMITIVES ARE THE SIMPLEST DATA TYPES IN JAVASCRIPT AND ARE IMMUTABLE. OBJECTS ARE MORE COMPLEX DATA TYPES THAT ARE USED TO REPRESENT REAL-WORLD OBJECTS OR ABSTRACT CONCEPTS AND ARE MUTABLE. IT IS IMPORTANT TO UNDERSTAND THE DIFFERENCES BETWEEN THESE TWO TYPES OF DATA IN ORDER TO WRITE EFFECTIVE AND MAINTAINABLE CODE IN JAVASCRIPT.

OPERATORS AND EXPRESSIONS

OPERATORS IN JAVASCRIPT ARE SYMBOLS THAT PERFORM SPECIFIC OPERATIONS ON ONE OR MORE OPERANDS (VALUES OR VARIABLES). FOR EXAMPLE, THE ADDITION OPERATOR (+) ADDS TWO OPERANDS TOGETHER AND THE ASSIGNMENT OPERATOR (=) ASSIGNS A VALUE TO A VARIABLE.
THERE ARE SEVERAL TYPES OF OPERATORS IN JAVASCRIPT, INCLUDING:

- ARITHMETIC OPERATORS (E.G. +, -, *, /, %)
- COMPARISON OPERATORS (E.G. >, <, >=, <=, ==, !=)
- LOGICAL OPERATORS (E.G. &&, ||, !)
- ASSIGNMENT OPERATORS (E.G. =, +=, -=, *=, /=)
- CONDITIONAL (TERNARY) OPERATOR (E.G. ?:)

EXPRESSIONS ARE COMBINATIONS OF VALUES, VARIABLES, AND OPERATORS THAT PRODUCE A RESULT. FOR EXAMPLE:

```
let x = 10;
let y = 20;
let z = x + y; // z is 30
```

IN THIS EXAMPLE, THE EXPRESSION "X + Y" IS EVALUATED TO 30 AND ASSIGNED TO THE "Z" VARIABLE.

OPERATOR PRECEDENCE DETERMINES THE ORDER IN WHICH OPERATORS ARE APPLIED WHEN AN EXPRESSION HAS MULTIPLE OPERATORS. FOR EXAMPLE:

```
let x = 10 + 5 * 3; // x is 25
```

IN THIS EXAMPLE, THE MULTIPLICATION OPERATOR (*) HAS A HIGHER PRECEDENCE THAN THE ADDITION OPERATOR (+), SO THE MULTIPLICATION IS PERFORMED BEFORE THE ADDITION. AS A RESULT, THE EXPRESSION IS EVALUATED AS 10 + (5 * 3) = 25.

YOU CAN USE PARENTHESES TO SPECIFY THE ORDER OF OPERATIONS IN AN EXPRESSION. FOR EXAMPLE:

```
let x = (10 + 5) * 3; // x is 45
```

IN THIS EXAMPLE, THE PARENTHESES INDICATE THAT THE ADDITION SHOULD BE PERFORMED BEFORE THE MULTIPLICATION. AS A RESULT, THE EXPRESSION IS EVALUATED AS (10 + 5) * 3 = 45.

CONCLUSION

In summary, operators are symbols that perform specific operations on one or more operands, and expressions are combinations of values, variables, and operators that produce a result. Operator precedence determines the order in which operators are applied in an expression, and parentheses can be used to specify the order of operations. Understanding how to use operators and expressions is an important part of programming in JavaScript.

VAR VS LET VS CONST

In JavaScript, there are three ways to declare variables: var, let, and const. Each of these keywords has its own rules and uses, and it is important to understand the differences between them in order to write effective and maintainable code.

VAR

The "var" keyword is used to declare variables in JavaScript. It was introduced in the early days of the language and was the only way to declare variables for a long time. However, the "var" keyword has some limitations and has been largely replaced by the "let" and "const" keywords in modern JavaScript.

One of the main issues with "var" is that it is function-scoped, rather than block-scoped. This means that variables declared with "var" are accessible within the entire function in which they are declared, rather than just within the block of code in which they appear. This can lead to unexpected behavior and can make it difficult to reason about the scope of variables in your code.

LET

The "let" keyword was introduced in ECMAScript 6 (also known as ES6) and is used to declare variables that can be reassigned later. "let" variables are block-scoped, which means that they are only accessible within the block of code in which they are declared. This makes them more predictable and easier to reason about than "var" variables.

For example:

```
if (x > 10) {let y = 20;
  console.log(y); // 20}
console.log(y); // ReferenceError: y is not defined
```

In this example, the "y" variable is declared with the "let" keyword and is only accessible within the block of the if statement. If you try to access it outside of the block, you will get a "ReferenceError" because "y" is not defined in that scope.

CONST

The "const" keyword was also introduced in ES6 and is used to declare variables that cannot be reassigned later. "const" variables are also block-scoped and behave similarly to "let" variables in that respect. However, the main difference is that "const" variables must be initialized with a value when they are declared and cannot be reassigned later.

For example:

```
const PI = 3.14;
PI = 3.14159; // TypeError: Assignment to constant variable.
```

JAVASCRIPT BASICS

IF ELSE CONDITIONALS

THE "IF" STATEMENT IN JAVASCRIPT IS USED TO EXECUTE A BLOCK OF CODE IF A CERTAIN CONDITION IS MET. THE "ELSE" CLAUSE IS USED TO EXECUTE A BLOCK OF CODE IF THE CONDITION IS NOT MET.

HERE IS THE BASIC SYNTAX FOR AN "IF" STATEMENT:

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

Here is the syntax for an "if" statement with an "else" clause:

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

THE CONDITION IS A BOOLEAN EXPRESSION THAT EVALUATES TO EITHER TRUE OR FALSE. IF THE CONDITION IS TRUE, THE CODE IN THE "IF" BLOCK IS EXECUTED. IF THE CONDITION IS FALSE, THE CODE IN THE "ELSE" BLOCK IS EXECUTED (IF PRESENT).

FOR EXAMPLE:

```
let x = 10;if (x > 5) {
  console.log("x is greater than 5");} else {
  console.log("x is not greater than 5");}
```

In this example, the condition "x > 5" is true, so the code in the "if" block is executed and the message "x is greater than 5" is printed to the console

IF ELSE LADDER

THE "IF-ELSE LADDER" IS A CONTROL STRUCTURE IN JAVASCRIPT THAT ALLOWS YOU TO EXECUTE A DIFFERENT BLOCK OF CODE DEPENDING ON MULTIPLE CONDITIONS. IT IS CALLED A LADDER BECAUSE IT CONSISTS OF MULTIPLE "IF" AND "ELSE" STATEMENTS ARRANGED IN A LADDER-LIKE FASHION.

HERE IS THE SYNTAX FOR AN "IF-ELSE LADDER":

```
if (condition1)
{// code to be executed if condition1 is true} else if (condition2)
{// code to be executed if condition1 is false and condition2 is true}
else if (condition3)
{// code to be executed if condition1 and condition2 are false and condition3 is true} ...
else {// code to be executed if all conditions are false}
```

In this structure, each "if" statement is followed by an optional "else" statement. If the first "if" condition is true, the code in the corresponding block is executed and the rest of the ladder is skipped. If the first "if" condition is false, the second "if" condition is evaluated, and so on. If none of the conditions are true, the code in the "else" block is executed.

For example:

```
let x = 10;
if (x > 15){
  console.log("x is greater than 15");
} else if (x > 10) {
  console.log("x is greater than 10 but less than or equal to 15");
} else if (x > 5) {
  console.log("x is greater than 5 but less than or equal to 10");
} else {
  console.log("x is less than or equal to 5");
}
```

In this example, the first "if" condition "x > 15" is false, so the second "if" condition "x > 10" is evaluated. This condition is also false, so the third "if" condition "x > 5" is evaluated. This condition is true, so the code in the corresponding block is executed and the message "x is greater than 5 but less than or equal to 10" is printed to the console.

The "if-else ladder" is a useful control structure for executing different blocks of code based on multiple conditions. It can help you write more concise and maintainable code in JavaScript.

SWITCH CASE

The "switch" statement in JavaScript is another control structure that allows you to execute a different block of code depending on a specific value. It is often used as an alternative to the "if-else ladder" when you have multiple conditions to check against a single value.

Here is the syntax for a "switch" statement:

```
switch (expression) {
  case value1:
    // code to be executed if expression == value1
    break;
  case value2:
    // code to be executed if expression == value2
    break;
  ...
  default:
    // code to be executed if expression does not match any of the values
}
```

In this structure, the "expression" is evaluated and compared to each of the "case" values. If the "expression" matches a "case" value, the corresponding block of code is executed. The "break" statement is used to exit the "switch" statement and prevent the code in the following cases from being executed. The "default" case is optional and is executed if the "expression" does not match any of the "case" values.

For example:

```
let x = "apple";
switch (x) {
  case "apple":
    console.log("x is an apple");
    break;
```

```
 case "banana":
 console.log("x is a banana");
 break;
 case "orange":
 console.log("x is an orange");
 break;
 default:
 console.log("x is something else");
}
```

In this example, the "expression" is the variable "x," which has the value "apple." The "expression" is compared to each of the "case" values, and when it matches the value "apple," the corresponding block of code is executed and the message "x is an apple" is printed to the console.

The "switch" statement is a useful control structure for executing different blocks of code based on a specific value. It can help you write more concise and maintainable code in JavaScript.

TERNARY OPERATOR

The ternary operator is a shorthand way to write an if-else statement in JavaScript. It takes the form of condition ? value1 : value2, where condition is a boolean expression, and value1 and value2 are expressions of any type. If condition is true, the ternary operator returns value1; if condition is false, it returns value2.

Here's an example of how you can use the ternary operator to assign a value to a variable based on a condition:

```
let x = 10;
let y = 20;
let max;
max = (x > y) ? x : y;
console.log(max); // Outputs: 20
```

In this example, the ternary operator checks whether x is greater than y. If it is, max is assigned the value of x; otherwise, it is assigned the value of y.

The ternary operator can be a useful and concise way to write simple if-else statements, but it can become difficult to read and understand when used for more complex statements or nested inside other expressions. In these cases, it may be better to use a regular if-else statement instead.

FOR LOOPS

For loops are a common control flow structure in programming that allows you to repeat a block of code a specific number of times. In JavaScript, there are three types of for loops: the standard for loop, the for-in loop, and the for-of loop.

STANDARD FOR LOOP

The standard for loop has the following syntax:

```
for (initialization; condition; increment/decrement) {
  // code to be executed
}
```

The initialization the statement is executed before the loop starts and is typically used to initialize a counter variable. The condition is checked at the beginning of each iteration and if it is true, the loop continues.

If it is false, the loop exits. The increment/decrement statement is executed at the end of each iteration and is used to update the counter variable.

Here's an example of a standard for loop that counts from 1 to 10:

```
for (let i = 1; i <= 10; i++) {
  console.log(i);
}
```

This loop will print the numbers 1 through 10 to the console.

FOR-IN LOOP

The for-in loop is used to iterate over the properties of an object. It has the following syntax:

```
for (variable in object) {
  // code to be executed
}
```

The variable is assigned the name of each property in the object as the loop iterates over them.

Here's an example of a for-in loop that iterates over the properties of an object:

```
let person = {
  name: "John",
  age: 30,
  job: "developer"
};
for (let key in person) {
  console.log(key + ": " + person[key]);}
```

This loop will print the following to the console:
name: John
age: 30
job: developer

FOR-OF LOOP

The for-of loop is used to iterate over the values of an iterable object, such as an array or a string. It has the following syntax:

```
for (variable of object) {
  // code to be executed
}
```

The variable is assigned the value of each element in the object as the loop iterates over them.
Here's an example of a for-of loop that iterates over the elements of an array:

```
let numbers = [1, 2, 3, 4, 5];

for (let number of numbers) {
  console.log(number);}
```

This loop will print the numbers 1 through 5 to the console.
For loops are a powerful tool in JavaScript and can be used to perform a variety of tasks, such as iterating over arrays and objects, repeating a block of code a specific number of times, and more. With the three types of for loops available in JavaScript, you can choose the one that best fits your needs and use it to write more efficient and effective code.

WHILE LOOP

While loops are a control flow structure in programming that allow you to repeat a block of code while a certain condition is true. In JavaScript, the syntax for a while loop is:

```
while (condition) {
  // code to be executed
}
```

The condition is checked at the beginning of each iteration and if it is true, the code block is executed. If it is false, the loop exits.

Here's an example of a while loop that counts from 1 to 10:

```
let i = 1;
while (i <= 10) {
  console.log(i);
  i++;
}
```

This loop will print the numbers 1 through 10 to the console.

It's important to include a way to update the condition within the loop, otherwise it will become an infinite loop and will run forever. In the example above, the i++ statement increments the value of i by 1 at the end of each iteration, which eventually causes the condition to be false and the loop to exit.

While loops can be useful when you don't know exactly how many times you need to execute a block of code. For example, you might use a while loop to keep prompting a user for input until they provide a valid response.

```
let input = "";

while (input !== "yes" && input !== "no") {
  input = prompt("Please enter 'yes' or 'no':");
}
```

This loop will keep prompting the user for input until they enter either "yes" or "no".

While loops can be a useful tool in JavaScript, but it's important to use them with caution. If the condition is never met, the loop will become an infinite loop and will run forever. Make sure to include a way to update the condition and eventually exit the loop to avoid this issue.

FUNCTIONS

JavaScript functions are blocks of code that can be defined and executed whenever needed. They are a crucial part of JavaScript programming and are used to perform specific tasks or actions.

Functions are often referred to as "first-class citizens" in JavaScript because they can be treated like any other value, such as a number or a string. This means that they can be assigned to variables, passed as arguments to other functions, and returned as values from functions.

Here's the basic syntax for defining a function in JavaScript:

```
function functionName(parameters) {
  // code to be executed
}
```

The functionName is a unique identifier for the function, and the parameters are the variables that are passed to the function when it is called. These parameters act as placeholders for the actual values that are passed to the function when it is executed.

Here's an example of a simple function that takes a single parameter and returns the square of that number:

```
function square(x) {
  return x * x;
}
```

To call this function, you would simply use the function name followed by the arguments in parentheses:

```
let result = square(5);  // returns 25
```

Functions can also have multiple parameters, like this:

```
function add(x, y) {
  return x + y;
}
```

In this case, the add function takes two parameters, x and y, and returns their sum.

JavaScript also has a special type of function called an "arrow function," which uses a shorter syntax. Here's the same square a function defined using an arrow function:

```
const square = (x) => {
  return x * x;
};
```

Arrow functions are often used when you want to create a small, one-line function that doesn't require a separate function keyword.

Functions can be defined inside other functions, which is known as "nesting." This is useful for creating smaller, reusable blocks of code that can be called from within the larger function.

```
function outerFunction(x) {
  function innerFunction() {
    // code to be executed
  }
  // more code
}
```

In this example, the innerFunction is defined inside the outerFunction and can only be called from within that function.

In addition to these basic concepts, there are many other things you can do with functions in JavaScript, such as passing functions as arguments, creating anonymous functions, and using higher-order functions. These advanced techniques can make your code more efficient and flexible, and are essential tools for any JavaScript developer.

JAVASCRIPT OBJECTS

STRINGS

ONE OF THE MOST IMPORTANT ASPECTS OF JAVASCRIPT IS ITS ABILITY TO MANIPULATE STRINGS, WHICH ARE SEQUENCES OF CHARACTERS. IN THIS BLOG POST, WE WILL EXPLORE THE BASICS OF JAVASCRIPT STRINGS AND THE VARIOUS STRING METHODS THAT CAN BE USED TO MANIPULATE THEM.

A STRING IN JAVASCRIPT IS A SEQUENCE OF CHARACTERS ENCLOSED IN EITHER SINGLE OR DOUBLE QUOTES. FOR EXAMPLE, THE FOLLOWING ARE VALID STRINGS IN JAVASCRIPT:

"Hello World"

'Hello World'

JAVASCRIPT PROVIDES A NUMBER OF BUILT-IN METHODS FOR MANIPULATING STRINGS. SOME OF THE MOST COMMONLY USED STRING METHODS ARE:

LENGTH - THIS METHOD RETURNS THE NUMBER OF CHARACTERS IN A STRING. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN 11:

```
var str = "Hello World";
console.log(str.length);
```

CONCAT - THIS METHOD IS USED TO CONCATENATE (COMBINE) TWO OR MORE STRINGS. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN "HELLO WORLD":

```
var str1 = "Hello";
var str2 = " World";
console.log(str1.concat(str2));
```

INDEXOF – THIS METHOD IS USED TO FIND THE INDEX OF A SPECIFIC CHARACTER OR SUBSTRING IN A STRING. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN 6:

```
var str = "Hello World";
console.log(str.indexOf("W"));
```

SLICE – THIS METHOD IS USED TO EXTRACT A PORTION OF A STRING. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN "WORLD":

```
var str = "Hello World";
console.log(str.slice(6));
```

REPLACE – THIS METHOD IS USED TO REPLACE A SPECIFIC CHARACTER OR SUBSTRING IN A STRING. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN "HELLO UNIVERSE":

```
var str = "Hello World";
console.log(str.slice(6));
```

TOUPPERCASE AND TOLOWERCASE – THESE METHODS ARE USED TO CONVERT A STRING TO UPPERCASE OR LOWERCASE LETTERS. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN "HELLO WORLD" AND "HELLO WORLD" RESPECTIVELY:

```
var str = "Hello World";
console.log(str.toUpperCase());
console.log(str.toLowerCase());
```

THESE ARE JUST A FEW OF THE MANY STRING METHODS AVAILABLE IN JAVASCRIPT. BY UNDERSTANDING THE BASICS OF STRINGS AND THE VARIOUS METHODS THAT CAN BE USED TO MANIPULATE THEM, YOU CAN CREATE MORE DYNAMIC AND INTERACTIVE WEB PAGES. SO, START EXPERIMENTING WITH DIFFERENT STRING METHODS AND SEE WHAT YOU CAN CREATE!

ARRAYS AND ARRAY METHODS

ONE OF THE MOST IMPORTANT DATA STRUCTURES IN JAVASCRIPT IS THE ARRAY, WHICH IS A COLLECTION OF ELEMENTS. IN THIS BLOG POST, WE WILL EXPLORE THE BASICS OF JAVASCRIPT ARRAYS AND THE VARIOUS ARRAY METHODS THAT CAN BE USED TO MANIPULATE THEM.

AN ARRAY IN JAVASCRIPT IS A COLLECTION OF ELEMENTS ENCLOSED IN SQUARE BRACKETS. ELEMENTS CAN BE OF ANY DATA TYPE, INCLUDING NUMBERS, STRINGS, AND OTHER ARRAYS. FOR EXAMPLE, THE FOLLOWING IS A VALID ARRAY IN JAVASCRIPT:

```
var myArray = [1, "Hello", [2, 3]];
```

JAVASCRIPT PROVIDES A NUMBER OF BUILT-IN METHODS FOR MANIPULATING ARRAYS. SOME OF THE MOST COMMONLY USED ARRAY METHODS ARE:

LENGTH - THIS METHOD RETURNS THE NUMBER OF ELEMENTS IN AN ARRAY. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN 3:

```
var myArray = [1, "Hello", [2, 3]];
console.log(myArray.length);
```

PUSH - THIS METHOD IS USED TO ADD AN ELEMENT TO THE END OF AN ARRAY. FOR EXAMPLE, THE FOLLOWING CODE WILL ADD THE ELEMENT "WORLD" TO THE END OF THE ARRAY:

```
var myArray = [1, "Hello", [2, 3]];
myArray.push("World");
console.log(myArray); // [1, "Hello", [2, 3], "World"]
```

POP – THIS METHOD IS USED TO REMOVE THE LAST ELEMENT OF AN ARRAY. FOR EXAMPLE, THE FOLLOWING CODE WILL REMOVE THE LAST ELEMENT ("WORLD") FROM THE ARRAY:

```
var myArray = [1, "Hello", [2, 3], "World"];
myArray.pop();
console.log(myArray); // [1, "Hello", [2, 3]]
```

SHIFT – THIS METHOD IS USED TO REMOVE THE FIRST ELEMENT OF AN ARRAY. FOR EXAMPLE, THE FOLLOWING CODE WILL REMOVE THE FIRST ELEMENT (1) FROM THE ARRAY:

```
var myArray = [1, "Hello", [2, 3]];
myArray.shift();
console.log(myArray); // ["Hello", [2, 3]]
```

SLICE – THIS METHOD IS USED TO EXTRACT A PORTION OF AN ARRAY. FOR EXAMPLE, THE FOLLOWING CODE WILL EXTRACT THE ELEMENTS FROM INDEX 1 TO 2 (EXCLUSIVE):

```
var myArray = [1, "Hello", [2, 3]];
console.log(myArray.slice(1, 2)); // ["Hello"]
```

SPLICE - THIS METHOD IS USED TO ADD OR REMOVE ELEMENTS FROM AN ARRAY. FOR EXAMPLE, THE FOLLOWING CODE WILL REMOVE THE ELEMENT AT INDEX 1 AND ADD THE ELEMENTS "HELLO WORLD" AND [4, 5] AT INDEX 1:

```
var myArray = [1, "Hello", [2, 3]];
myArray.splice(1, 1, "Hello World", [4, 5]);
console.
```

LOOPS WITH ARRAYS

ONE OF THE MOST IMPORTANT DATA STRUCTURES IN JAVASCRIPT IS THE ARRAY, WHICH IS A COLLECTION OF ELEMENTS. WHEN WORKING WITH ARRAYS, IT IS OFTEN NECESSARY TO ITERATE THROUGH EACH ELEMENT IN THE ARRAY, WHICH IS WHERE LOOPS COME IN. IN THIS BLOG POST, WE WILL EXPLORE HOW TO USE LOOPS WITH ARRAYS IN JAVASCRIPT.

JAVASCRIPT PROVIDES SEVERAL WAYS TO ITERATE THROUGH AN ARRAY, INCLUDING THE FOR LOOP, FOREACH METHOD, AND FOR...OF LOOP.

FOR LOOP - THIS IS THE MOST BASIC WAY TO ITERATE THROUGH AN ARRAY. THE FOR LOOP USES A COUNTER VARIABLE THAT IS INCREMENTED ON EACH ITERATION. FOR EXAMPLE, THE FOLLOWING CODE WILL PRINT OUT EACH ELEMENT IN THE ARRAY:

```
var myArray = [1, 2, 3, 4, 5];
for (var i = 0; i < myArray.length; i++) {
    console.log(myArray[i]);
}
```

FOREACH METHOD – THIS METHOD IS A MORE CONCISE WAY TO ITERATE THROUGH AN ARRAY. THE FOREACH METHOD TAKES A CALLBACK FUNCTION AS ITS ARGUMENT, WHICH IS CALLED ON EACH ELEMENT IN THE ARRAY. FOR EXAMPLE, THE FOLLOWING CODE WILL PRINT OUT EACH ELEMENT IN THE ARRAY:

```
var myArray = [1, 2, 3, 4, 5];
myArray.forEach(function(element) {
    console.log(element);
});
VAR MYARRAY = [1, 2, 3, 4, 5];
MYARRAY.FOREACH(FUNCTION(ELEMENT) {
    CONSOLE.LOG(ELEMENT);
});
```

FOR...OF LOOP – THIS IS A MORE RECENT ADDITION TO JAVASCRIPT, AND IT IS THE MOST CONCISE WAY TO ITERATE THROUGH AN ARRAY. IT ALLOWS YOU TO ITERATE THROUGH THE ELEMENTS OF AN ARRAY WITHOUT HAVING TO ACCESS THE INDEX, AND IT WORKS WITH ANY ITERABLE OBJECT, NOT JUST ARRAYS. FOR EXAMPLE, THE FOLLOWING CODE WILL PRINT OUT EACH ELEMENT IN THE ARRAY:

```
var myArray = [1, 2, 3, 4, 5];
for (var element of myArray) {
    console.log(element);
}
```

IT IS IMPORTANT TO NOTE THAT WHEN YOU ARE ITERATING THROUGH AN ARRAY USING A FOR LOOP AND YOU PLAN TO CHANGE THE ARRAY DURING ITERATION YOU SHOULD USE A FOR LOOP WITH A SEPARATE COUNTER VARIABLE.

MAP, FILTER AND REDUCE

ONE OF THE MOST IMPORTANT DATA STRUCTURES IN JAVASCRIPT IS THE ARRAY, WHICH IS A COLLECTION OF ELEMENTS. WHEN WORKING WITH ARRAYS, IT IS OFTEN NECESSARY TO MANIPULATE THE ELEMENTS IN VARIOUS WAYS, WHICH IS WHERE THE BUILT-IN METHODS MAP, FILTER, AND REDUCE COME IN. IN THIS BLOG POST, WE WILL EXPLORE HOW TO USE THESE METHODS TO MANIPULATE ARRAYS IN JAVASCRIPT.

MAP - THE MAP METHOD IS USED TO CREATE A NEW ARRAY WITH THE RESULT OF A CALLBACK FUNCTION CALLED ON EACH ELEMENT IN THE ORIGINAL ARRAY. THE CALLBACK FUNCTION TAKES ONE ARGUMENT, THE CURRENT ELEMENT, AND RETURNS THE NEW VALUE FOR THAT ELEMENT. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN AN ARRAY OF EACH ELEMENT MULTIPLIED BY 2:

```
var myArray = [1, 2, 3, 4, 5];var multipliedArray = myArray.map(function(element) {return element * 2;});
console.log(multipliedArray); // [2, 4, 6, 8, 10]
```

FILTER – THE FILTER METHOD IS USED TO FILTER AN ARRAY BASED ON A CERTAIN CONDITION. THE FILTER METHOD TAKES A CALLBACK FUNCTION AS ITS ARGUMENT, WHICH IS CALLED ON EACH ELEMENT IN THE ARRAY. IF THE FUNCTION RETURNS TRUE, THE ELEMENT IS INCLUDED IN THE NEW FILTERED ARRAY. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN AN ARRAY OF ALL EVEN NUMBERS IN THE ORIGINAL ARRAY:

```
var myArray = [1, 2, 3, 4, 5];
var multipliedArray = myArray.map(function(element) {
return element * 2;});
console.log(multipliedArray); // [2, 4, 6, 8, 10]
```

REDUCE – THE REDUCE METHOD IS USED TO REDUCE AN ARRAY TO A SINGLE VALUE. THE REDUCE METHOD TAKES A CALLBACK FUNCTION AS ITS ARGUMENT, WHICH IS CALLED ON EACH ELEMENT IN THE ARRAY. THE CALLBACK FUNCTION TAKES TWO ARGUMENTS, THE ACCUMULATOR AND THE CURRENT ELEMENT, AND RETURNS THE NEW VALUE FOR THE ACCUMULATOR. FOR EXAMPLE, THE FOLLOWING CODE WILL RETURN THE SUM OF ALL ELEMENTS IN THE ARRAY:

```
var myArray = [1, 2, 3, 4, 5];
var sum = myArray.reduce(function(acc, cur) {
    return acc + cur;
}, 0);
console.log(sum); // 15
```

IT'S IMPORTANT TO NOTE THAT THESE METHODS WORK ON THE ORIGINAL ARRAY AND DOESN'T CHANGE IT, IT RETURNS A NEW ARRAY.

COMBINING THESE THREE METHODS CAN BE POWERFUL, FOR EXAMPLE, YOU CAN FILTER AN ARRAY, THEN MAP THE FILTERED ARRAY, AND FINALLY REDUCE THE MAPPED ARRAY.
IN CONCLUSION, UNDERSTANDING AND UTILIZING THE MAP, FILTER, AND REDUCE METHODS IN JAVASCRIPT CAN GREATLY IMPROVE YOUR ABILITY TO MANIPULATE ARRAYS AND PERFORM COMPLEX OPERATIONS ON THEM. THESE METHODS ARE NOT ONLY MORE CONCISE BUT ALSO MORE EFFICIENT THAN TRADITIONAL LOOPS AND MAKE YOUR CODE MORE READABLE.

DATE

JAVASCRIPT'S DATE OBJECT ALLOWS YOU TO WORK WITH DATES AND TIMES IN YOUR SCRIPTS. IT CAN BE USED TO GET THE CURRENT DATE AND TIME, OR TO MANIPULATE AND FORMAT DATES AND TIMES IN VARIOUS WAYS.
CREATING A NEW DATE OBJECT IS SIMPLE. YOU CAN EITHER CREATE A NEW DATE OBJECT WITH THE CURRENT DATE AND TIME BY CALLING NEW DATE() WITH NO ARGUMENTS, OR YOU CAN CREATE A DATE OBJECT WITH A SPECIFIC DATE AND TIME BY PASSING IN A STRING OR NUMERICAL VALUES.
HERE'S AN EXAMPLE OF HOW YOU CAN USE THE DATE OBJECT TO GET THE CURRENT DATE AND TIME:

```
var currentDate = new Date();
console.log(currentDate);
```

OUTPUT: THU JAN 14 2021 15:15:38 GMT+0530 (INDIA STANDARD TIME)

YOU CAN ALSO MANIPULATE THE DATE AND TIME USING THE VARIOUS METHODS AVAILABLE ON THE DATE OBJECT. FOR EXAMPLE, YOU CAN USE THE SETDATE() METHOD TO SET THE DAY OF THE MONTH, AND THE SETFULLYEAR() METHOD TO SET THE YEAR.

```
VAR DATE = NEW DATE();
DATE.SETDATE(15);
DATE.SETFULLYEAR(2022);
CONSOLE.LOG(DATE);
```

OUTPUT: SAT JAN 15 2022 15:15:38 GMT+0530 (INDIA STANDARD TIME)

YOU CAN ALSO FORMAT THE DATE AND TIME USING THE TOLOCALESTRING() METHOD, WHICH ALLOWS YOU TO SPECIFY THE FORMAT AND TIME ZONE.

```
var date = new Date();
console.log(date.toLocaleString());
console.log(date.toLocaleString('en-US', {timeZone: 'UTC'}));
```

OUTPUT: 1/14/2021, 3:45:38 PM 1/14/2021, 10:15:38 AM

IN THIS WAY, YOU CAN USE THE DATE OBJECT IN JAVASCRIPT TO WORK WITH DATES AND TIMES IN YOUR SCRIPTS, AND TO FORMAT AND MANIPULATE DATES AND TIMES IN VARIOUS WAYS.

MATH

The JavaScript Math object is a built-in object that provides a variety of mathematical functions and constants. It can be used to perform mathematical operations such as trigonometry, logarithms, and random number generation.

One of the most commonly used functions in the Math object is the Math.random() function, which generates a random number between 0 (inclusive) and 1 (exclusive). This function can be used to generate random numbers for various purposes such as games, simulations, and other applications.

```
console.log(Math.random());
// Output: a random number between 0 and 1 (e.g. 0.3456)
```

You can also use the Math.floor() function to round a number down to the nearest integer.

```
console.log(Math.floor(3.8));
// Output: 3
```

The Math.ceil() function rounds a number up to the nearest integer.

```
console.log(Math.ceil(3.2));
// Output: 4
```

Another useful function of Math object is Math.max() and Math.min() which are used to find the maximum and minimum values in a set of numbers respectively.

```
console.log(Math.max(3,5,7,9));
// Output: 9
console.log(Math.min(3,5,7,9));
// Output: 3
```

Math object also provides the most commonly used mathematical constants such as Math.PI for the value of pi, Math.E for the value of the mathematical constant e.

```
console.log(Math.PI);
// Output: 3.141592653589793
console.log(Math.E);
// Output: 2.718281828459045
```

In this way, you can use the Math object in JavaScript to perform mathematical operations, generate random numbers, and access mathematical constants. It is a powerful tool that can be used to add more functionality to your JavaScript programs and make them more interactive and dynamic.

NUMBER

The JavaScript Number object is a built-in object that provides a number of properties and methods for working with numeric values. The Number object can be used to convert a value to a number, perform mathematical operations, and format numbers for display.

One of the most commonly used methods of the Number object is the Number() function, which can be used to convert a value to a number. This function can be used to convert strings, booleans, and other types of values to numbers.

```
console.log(Number("3.14"));
// Output: 3.14
console.log(Number(true));
// Output: 1
```

You can also use the parseInt() and parseFloat() methods to convert strings to integers and floating-point numbers, respectively.

```
console.log(parseInt("3"));
// Output: 3
console.log(parseFloat("3.14"));
// Output: 3.14
```

The Number object also provides several properties that can be used to access the maximum and minimum values that can be represented by a number in JavaScript. These properties are Number.MAX_VALUE and Number.MIN_VALUE.

```
console.log(Number.MAX_VALUE);
// Output: 1.7976931348623157e+308
console.log(Number.MIN_VALUE);
// Output: 5e-324
```

Another useful property of the Number object is Number.POSITIVE_INFINITY and Number.NEGATIVE_INFINITY which represents the positive and negative infinity respectively.

```
console.log(1 / 0);
// Output: Infinity
console.log(-1 / 0);
// Output: -Infinity
```

The Number object also provides several methods for formatting numbers for display, such as the toFixed() method, which can be used to format a number with a specific number of decimal places.

```
console.log((3.14159265358979323846264338327795).toFixed(2));
// Output: "3.14"
```

BOOLEAN

The JavaScript Boolean object is a built-in object that provides a way to work with Boolean (true/false) values in your scripts. The Boolean object can be used to create Boolean values and perform logical operations, such as and, or, and not.

A boolean value can be created by using the Boolean() function or by assigning a value of true or false to a variable.

```
var isTrue = Boolean(1);
console.log(isTrue); // Output: true
var isFalse = Boolean(0);
console.log(isFalse); // Output: false
```

You can also use the logical && operator to check if both expressions are true and the || operator to check if at least one of the expressions is true.

```
var x = true;
var y = false;
console.log(x && y); // Output: false
console.log(x || y); // Output: true
```

You can also use the ! operator to invert a boolean value.

```
console.log(!x); // Output: false
console.log(!y); // Output: true
```

DOM & BOM

WINDOW OBJECT

THE JAVASCRIPT WINDOW OBJECT REPRESENTS THE CURRENT BROWSER WINDOW OR TAB THAT IS OPEN IN A WEB BROWSER. IT IS A GLOBAL OBJECT THAT PROVIDES ACCESS TO VARIOUS PROPERTIES AND METHODS RELATED TO THE BROWSER WINDOW.

ONE OF THE MOST COMMONLY USED PROPERTIES OF THE WINDOW OBJECT IS THE DOCUMENT PROPERTY, WHICH REPRESENTS THE CURRENT WEB PAGE. THIS PROPERTY CAN BE USED TO ACCESS THE HTML ELEMENTS ON A PAGE, AS WELL AS MANIPULATE THEM. FOR EXAMPLE, THE FOLLOWING CODE CAN BE USED TO CHANGE THE TEXT OF A SPECIFIC ELEMENT ON A PAGE:

```
document.getElementById("myElement").innerHTML = "This is my new text";
```

ANOTHER IMPORTANT PROPERTY OF THE WINDOW OBJECT IS THE LOCATION PROPERTY, WHICH PROVIDES INFORMATION ABOUT THE CURRENT URL. THIS PROPERTY CAN BE USED TO REDIRECT THE USER TO A DIFFERENT PAGE, AS WELL AS RETRIEVE THE CURRENT URL. FOR EXAMPLE, THE FOLLOWING CODE CAN BE USED TO REDIRECT THE USER TO A DIFFERENT PAGE:

```
window.location.href = "https://www.example.com";
```

THE WINDOW OBJECT ALSO PROVIDES SEVERAL METHODS FOR DISPLAYING DIALOG BOXES, SUCH AS ALERT(), CONFIRM(), AND PROMPT(). THESE METHODS CAN BE USED TO DISPLAY MESSAGES TO THE USER AND RECEIVE INPUT FROM THE USER.

alert("This is an alert message");
There are many other properties and methods provided by the window object, such as setTimeout() and setInterval() for scheduling tasks, open() and close() for opening and closing new windows, and scrollTo() for scrolling the window to a specific position.
It's important to note that window object is also the parent object of the document object and history object.
In conclusion, the JavaScript window object provides a powerful set of tools for interacting with the browser window and web page. It can be used to access and manipulate HTML elements, redirect the user to a different page, and display dialog boxes. Understanding how to use the window object is essential for creating interactive and dynamic web pages.

HISTORY OBJECT

The JavaScript history object is a part of the window object and provides access to the browser's history. It allows developers to manipulate the browser's history and change the current URL without reloading the page.
One of the most commonly used methods of the history object is the pushState() method. This method allows developers to add a new entry to the browser's history and change the current URL without reloading the page. For example, the following code can be used to change the current URL to https://www.example.com/ and add an entry to the browser's history:
history.pushState(null, null, "https://www.example.com");

Another important method is the replaceState() method which is similar to pushState() but it replaces the current entry in the history instead of creating a new one.
It is also possible to navigate through the browser's history using the back() and forward() methods. These methods allow developers to navigate to the previous or next page in the history, respectively.
For example, the following code can be used to navigate to the previous page in the history:

```
history.back();
```

The length property of the history object returns the number of entries in the browser's history. Developers can use this property to check how many pages the user has visited and provide appropriate navigation options.
It's important to note that the history object is not a perfect solution for all navigation needs and it can only be used to navigate within the same origin. Also, the history object is part of the HTML5 specification and it is not supported in all browsers, so it's important to check the browser compatibility before using it.
In conclusion, the JavaScript history object provides a powerful set of tools for manipulating the browser's history and changing the current URL without reloading the page. Understanding how to use the history object is essential for creating single-page applications (SPA) and improving the user experience. However, it's important to keep in mind its limitations and check the browser compatibility before using it.

NAVIGATOR OBJECT

The JavaScript navigator object is a part of the window object and provides information about the browser and the device being used to access the web page.

It allows developers to gather information about the browser type, version, and supported features, as well as detect the device's properties such as its screen size, resolution, and online/offline status.
One of the most commonly used properties of the navigator object is the userAgent property. This property returns a string that represents the browser's user agent. Developers can use this property to detect the browser type and version. For example, the following code can be used to check if the user is using Google Chrome:

```
if (navigator.userAgent.indexOf("Chrome") !== -1) {
    console.log("The user is using Google Chrome");
}
```

Another important property is the onLine property which indicates whether the device is currently connected to the internet or not. This property can be useful for providing offline support or showing an offline message to the user.

```
if (!navigator.onLine) {
    console.log("The device is currently offline");
}
```

The navigator object also provides several methods for detecting the device's screen properties, such as screen.width and screen.height which returns the screen width and height in pixels. Developers can use these properties to create responsive designs and adjust the layout based on the screen size.

```
console.log(`The screen width is ${screen.width} pixels and the screen height is ${screen.height} pixels`);
```

The navigator object also provides several methods for detecting the device's Geolocation, such as navigator.geolocation.getCurrentPosition() which returns the position of the device based on its GPS coordinates or the IP address.

It's important to note that not all properties and methods provided by the navigator object are supported by all browsers and devices. Therefore, it's important to check browser compatibility before using them.

In conclusion, the JavaScript navigator object provides a powerful set of tools for gathering information about the browser and the device being used to access the web page. Understanding how to use the navigator object is essential for creating responsive designs and providing offline support, but it's important to keep in mind its limitations and check browser compatibility before using it.

SCREEN OBJECT

The JavaScript screen object is a part of the window object and provides information about the device's screen properties. It allows developers to gather information about the screen resolution, color depth, and available screen area.

One of the most commonly used properties of the screen object is the width and height properties. These properties return the width and height of the device's screen in pixels, respectively. Developers can use these properties to create responsive designs and adjust the layout based on the screen size.

For example, the following code can be used to log the screen resolution:

```
console.log(`The screen resolution is ${screen.width}x${screen.height} pixels`);
```

Another important property is the colorDepth property which returns the number of bits used to represent the color of a single pixel. This property can be used to check the color depth of the screen and make sure that images and videos are displayed correctly.

```
console.log(`The screen color depth is ${screen.colorDepth} bits`);
```

The screen object also provides the availWidth and availHeight properties, which return the width and height of the available screen area, respectively. This means the area available to the browser, not including the taskbar or other system interface elements.

```
console.log(`The available screen width is ${screen.availWidth} pixels and the available screen height is ${screen.availHeight} pixels`);
```

It's important to note that the screen object is a part of the DOM API and it is supported by all modern browsers, but it's important to check for browser compatibility before using it. Also, the screen object properties will return the screen size of the device the browser is running on, and not the size of the window that the browser is running in.

In conclusion, the JavaScript screen object provides a powerful set of tools for gathering information about the device's screen properties such as resolution, color depth and available screen area. Understanding how to use the screen object is essential for creating responsive designs and adjusting the layout based on the screen size, but it's important to keep in mind its limitations and check browser compatibility before using it.

DOCUMENT OBJECT

The JavaScript document object is a part of the window object and represents the current web page. It allows developers to access and manipulate the elements of the web page, as well as the Document Object Model (DOM) tree that represents the structure of the web page.

One of the most commonly used methods of the document object is the getElementById() method. This method allows developers to access a specific element on the web page by its unique ID. For example, the following code can be used to change the text of an element with the ID "myElement":

```
document.getElementById("myElement").innerHTML = "This is my new text";
```

Another important method is getElementsByClassName() method which returns a live HTMLCollection of elements with the given class name. It can be used to access multiple elements with the same class name.

```
let elements = document.getElementsByClassName("myClass");
for (let i = 0; i < elements.length; i++) {
    elements[i].innerHTML = "This is my new text";
}
```

The document object also provides several methods for creating new elements and adding them to the web page, such as createElement() and createTextNode(). For example, the following code can be used to create a new div element and add it to the web page:

```
let newDiv = document.createElement("div");
document.body.appendChild(newDiv);
```

It's important to note that the document object is a part of the DOM API and it is supported by all modern browsers, but it's important to check for browser compatibility before using it.
In conclusion, the JavaScript document object provides a powerful set of tools for accessing and manipulating the elements of the web page and the Document Object Model (DOM) tree.
Understanding how to use the document object is essential for creating dynamic and interactive web pages, but it's important to keep in mind its limitations and check browser compatibility before using it.

GETELEMENTBYID

The getElementById() method is a part of the JavaScript document object and it allows developers to access a specific element on a web page by its unique ID. This method returns the first element that matches the specified ID, or null if no such element is found.
For example, consider the following HTML code:

```
<div id="myDiv">This is my div</div>
```

The following JavaScript code can be used to access the element with the ID "myDiv" and change its text:

```
let myDiv = document.getElementById("myDiv");
myDiv.innerHTML = "This is my new text";
```

The getElementById() method is a convenient way to access a specific element on a web page, as it saves developers from having to traverse the entire DOM tree to find the element. This method is especially useful when working with large and complex web pages, as it allows developers to quickly and easily access the elements they need.

It's important to note that the ID of an element must be unique within the web page, as the getElementById() method only returns the first element that matches the specified ID. Also, the getElementById() method is case-sensitive, meaning that "myDiv" and "mydiv" are considered to be different IDs.

In conclusion, the getElementById() method is a powerful tool for accessing specific elements on a web page by their unique ID. Understanding how to use this method is essential for creating dynamic and interactive web pages, as it allows developers to quickly and easily access the elements they need. However, it's important to keep in mind that the ID of an element must be unique within the web page and the method is case-sensitive.

GETELEMENTSBYCLASSNAME

The getElementsByClassName() method is a part of the JavaScript document object and it allows developers to access multiple elements on a web page by their class name. This method returns a live HTMLCollection of elements with the given class name, or an empty HTMLCollection if no such elements are found.

For example, consider the following HTML code:

```
<div class="myClass">This is my div</div>
<div class="myClass">This is my div</div>
<div class="myClass">This is my div</div>
```

The following JavaScript code can be used to access the elements with the name attribute "myName" and change their value:

```
let elements = document.getElementsByName("myName");
for (let i = 0; i < elements.length; i++) {
    elements[i].value = "This is my new value";
}
```

The getElementsByName() method is a convenient way to access multiple elements on a web page, as it allows developers to select elements based on their name attribute, which is a common way to group elements with similar functionality. For example, radio buttons or checkboxes that are related to a certain topic should have the same name attribute. This method is especially useful when working with large and complex web pages, as it allows developers to quickly and easily access multiple elements that share the same name attribute.

It's important to note that the getElementsByName() method returns a live HTMLCollection, which means that the collection is updated automatically as elements are added or removed from the web page. Also, this method is case-sensitive, meaning that "myName" and "myname" are considered to be different name attributes.

In conclusion, the getElementsByName() method is a powerful tool for accessing multiple elements on a web page by their name attribute. Understanding how to use this method is essential for creating dynamic and interactive web pages, as it allows developers to quickly and easily access multiple elements that share the same name attribute. However, it's important to keep in mind that the method returns a live HTMLCollection and is case-sensitive. Additionally, this method is mostly used for form elements like radio buttons, checkboxes and input fields.

GETELEMENTSBYNAME

The getElementsByTagName() method is a part of the JavaScript document object and it allows developers to access multiple elements on a web page by their HTML tag name. This method returns a live HTMLCollection of elements with the given tag name, or an empty HTMLCollection if no such elements are found.

For example, consider the following HTML code:

```
<input type="text" name="myName" value="">
<input type="text" name="myName" value="">
<input type="text" name="myName" value="">
```

The following JavaScript code can be used to access the elements with the name attribute "myName" and change their value:

```
let elements = document.getElementsByName("myName");
for (let i = 0; i < elements.length; i++) {
    elements[i].value = "This is my new value";
}
```

The getElementsByName() method is a convenient way to access multiple elements on a web page, as it allows developers to select elements based on their name attribute, which is a common way to group elements with similar functionality.

For example, radio buttons or checkboxes that are related to a certain topic should have the same name attribute. This method is especially useful when working with large and complex web pages, as it allows developers to quickly and easily access multiple elements that share the same name attribute.
It's important to note that the getElementsByName() method returns a live HTMLCollection, which means that the collection is updated automatically as elements are added or removed from the web page. Also, this method is case-sensitive, meaning that "myName" and "myname" are considered to be different name attributes.
In conclusion, the getElementsByName() method is a powerful tool for accessing multiple elements on a web page by their name attribute. Understanding how to use this method is essential for creating dynamic and interactive web pages, as it allows developers to quickly and easily access multiple elements that share the same name attribute. However, it's important to keep in mind that the method returns a live HTMLCollection and is case-sensitive. Additionally, this method is mostly used for form elements like radio buttons, checkboxes and input fields.

GETELEMENTSBYNAME

The getElementsByTagName() method is a part of the JavaScript document object and it allows developers to access multiple elements on a web page by their HTML tag name. This method returns a live HTMLCollection of elements with the given tag name, or an empty HTMLCollection if no such elements are found.
For example, consider the following HTML code:

```
<p>This is my paragraph</p>
<p>This is my paragraph</p>
<p>This is my paragraph</p>
```

The following JavaScript code can be used to access the <p> elements on the web page and change their text:

```
let elements = document.getElementsByTagName("p");
for (let i = 0; i < elements.length; i++) {
    elements[i].innerHTML = "This is my new text";
}
```

The getElementsByTagName() method is a convenient way to access multiple elements on a web page, as it allows developers to select elements based on their HTML tag name. This method is especially useful when working with large and complex web pages, as it allows developers to quickly and easily access multiple elements of the same type.

It's important to note that the getElementsByTagName() method returns a live HTMLCollection, which means that the collection is updated automatically as elements are added or removed from the web page. Also, this method is case-sensitive, meaning that "p" and "P" are considered to be different tag names.

In conclusion, the getElementsByTagName() method is a powerful tool for accessing multiple elements on a web page by their HTML tag name. Understanding how to use this method is essential for creating dynamic and interactive web pages, as it allows developers to quickly and easily access multiple elements of the same type.

It's important to note that the getElementsByTagName() method returns a live HTMLCollection, which means that the collection is updated automatically as elements are added or removed from the web page. Also, this method is case-sensitive, meaning that "p" and "P" are considered to be different tag names.

In conclusion, the getElementsByTagName() method is a powerful tool for accessing multiple elements on a web page by their HTML tag name. Understanding how to use this method is essential for creating dynamic and interactive web pages, as it allows developers to quickly and easily access multiple elements of the same type. However, it's important to keep in mind that the method returns a live HTMLCollection and is case-sensitive. It's a good practice to use this method in combination with other methods like getElementById and getElementsByClassName to target specific elements on the web page.

INNERHTML

The innerHTML property is a part of the JavaScript HTMLElement object and it allows developers to access and manipulate the HTML content of an element. The innerHTML property returns the content between the opening and closing tags of an element, as a string of HTML.

For example, consider the following HTML code:

```
<div id="myDiv">
    <p>This is my paragraph</p>
    <p>This is my paragraph</p>
</div>
```

The following JavaScript code can be used to access the <p> elements on the web page and change their text:

```
let myDiv = document.getElementById("myDiv");
myDiv.innerHTML = "<p>This is my new text</p>";
```

The innerHTML property is a powerful tool for manipulating the HTML content of an element. Developers can use it to add, remove, or replace elements, as well as change the text and attributes of existing elements.

It's important to note that the innerHTML property can be used to insert any valid HTML code, including scripts and event handlers. This can be a powerful feature, but it can also pose a security risk if the HTML is not properly sanitized. Also, it's important to note that when you set the innerHTML property, the content is completely replaced, any previous content, event handlers, and other properties are lost.

In conclusion, the innerHTML property is a powerful tool for manipulating the HTML content of an element in JavaScript. Understanding how to use this property is essential for creating dynamic and interactive web pages, as it allows developers to easily add, remove, or replace elements, as well as change the text and attributes of existing elements. However, it's important to keep in mind that this property can pose a security risk if the HTML is not properly sanitized and also, it completely replaces the content when set.

OUTERHTML

The outerHTML property is a part of the JavaScript HTMLElement object and it allows developers to access and manipulate the entire HTML of an element, including the element's own tags.

The outerHTML property returns the entire HTML of an element, as a string of HTML.
For example, consider the following HTML code:

```
let myDiv = document.getElementById("myDiv");
myDiv.innerHTML = "<p>This is my new text</p>";
```

OOPS

CLASS

The outerHTML property returns the entire HTML of an element, as a string of HTML.
For example, consider the following HTML code:

```
class Person
{constructor(name, age) {
this.name = name;t
his.age = age;
}
}
```

The constructor method is a special method that is called when a new object is created from the class. It is used to initialize the properties of the object. In this example, the constructor takes in two parameters name and age and assigns them to the object's properties.
You can create a new object using the new keyword and passing in any required arguments to the constructor. For example:

```
let person1 = new Person("John", 30);
let person2 = new Person("Jane", 25);
```

The outerHTML property returns the entire HTML of an element, as a string of HTML. For example, consider the following HTML code:

```
class Person {
    constructor(name, age) {
        this.name = name;
        this.age = age;
    }

    greet() {
        return `Hello, my name is ${this.name} and I am ${this.age} years
old.`;
    }
}
```

JavaScript classes also support inheritance and polymorphism which are fundamental concepts of OOP, but beyond the scope of this blog.

In conclusion, classes in JavaScript are used to define the blueprint for objects and provide a way to create objects that share the same properties and methods. The class keyword is used to define a class, and the constructor method is used to initialize the properties of the object.

OBJECTS

Object-oriented programming (OOP) is a programming paradigm that uses objects and their interactions to design applications and computer programs. In JavaScript, objects are a fundamental concept, and they are used to represent real-world entities, such as a person, a car, or a bank account. JavaScript objects are created using the {} notation, also known as object literal notation. For example, the following code creates an object that represents a person:

```
let person = {
    name: "John",
    age: 30,
    greet: function() {
        return `Hello, my name is ${this.name} and I am ${this.age} years
old.`;
    }
};
```

You can access the properties of an object using the dot notation (.) or the bracket notation ([]). For example, you can access the name and age properties of the person object using:

```
console.log(person.name); // "John"
console.log(person["age"]); // 30
```

You can also add, update and delete properties to an object by using the dot notation or the bracket notation.

```
person.address = "New York";
person["phone"] = "123-456-7890";
delete person.age;
```

JavaScript objects also have built-in methods such as hasOwnProperty(), valueOf(), toString() etc.

Objects in JavaScript can also be created using a constructor function, which is a special kind of function that is used to create and initialize new objects. For example, the following code defines a constructor function Person that can be used to create new Person objects:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        return `Hello, my name is ${this.name} and I am ${this.age} years old.`;
    }
}
let person1 = new Person("John", 30);
let person2 = new Person("Jane", 25);
```

In addition, JavaScript also provides a Object() constructor function that can be used to create an empty object with no properties.

In conclusion, objects in JavaScript are a fundamental concept that are used to represent real-world entities. They can be created using object literal notation or constructor functions, and they have properties and methods that can be accessed and manipulated.

CONSTRUCTOR

The constructor is a special method that is used in object-oriented programming (OOP) to create and initialize new objects. In JavaScript, constructors are used to create objects that share the same properties and methods, and they are defined using the constructor keyword.

A constructor function is defined using the function keyword, followed by the function name, which is typically the name of the class. The constructor function takes in any required parameters and assigns them to the object's properties.

For example, consider the following constructor function Person that defines a person's name and age:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```

To create a new object using the constructor function, you use the new keyword followed by the function name and any required parameters:

```
let person1 = new Person("John", 30);
let person2 = new Person("Jane", 25);
```

The this keyword inside the constructor function refers to the current object being created. The this keyword is used to assign the properties passed in as arguments to the new object being created.

In addition to initializing properties, constructors can also define methods for objects created from the constructor. For example, you can add a method called greet() that returns a greeting message:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
    this.greet = function() {
        return `Hello, my name is ${this.name} and I am ${this.age} years old.`;
```

STATIC METHOD

In object-oriented programming, a static method is a method that is associated with a class, rather than an instance of the class. In JavaScript, static methods are defined using the static keyword and they can be accessed without creating an instance of the class.

For example, consider the following class MathHelper that defines a static method called add() that takes in two numbers and returns their sum:

```
class MathHelper {
    static add(a, b) {
        return a + b;
    }
}
```

You can access the static method directly from the class, without creating an instance of the class.

```
console.log(MathHelper.add(1, 2)); // 3
```

CONSTRUCTOR

The constructor is a special method that is used in object-oriented programming (OOP) to create and initialize new objects. In JavaScript, constructors are used to create objects that share the same properties and methods, and they are defined using the constructor keyword.

A constructor function is defined using the function keyword, followed by the function name, which is typically the name of the class. The constructor function takes in any required parameters and assigns them to the object's properties.

For example, consider the following constructor function Person that defines a person's name and age:

```
function Person(name, age) {
  this.name = name;
  this.age = age;
}
```

To create a new object using the constructor function, you use the new keyword followed by the function name and any required parameters:

```
let person1 = new Person("John", 30);
let person2 = new Person("Jane", 25);
```

The this keyword inside the constructor function refers to the current object being created. The this keyword is used to assign the properties passed in as arguments to the new object being created.

In addition to initializing properties, constructors can also define methods for objects created from the constructor. For example, you can add a method called greet() that returns a greeting message:

ENCAPSULATION

Encapsulation is a fundamental concept in object-oriented programming (OOP) that refers to the practice of hiding the internal details of an object from other objects and code. In JavaScript, encapsulation is achieved by using closures and access modifiers, such as the private and public keywords.

JavaScript uses closures to implement encapsulation, which allows developers to create private variables and methods that cannot be accessed from outside the object. For example, consider the following code:

```
function Person(name, age) {
    let privateAge = age;
    this.name = name;

    this.getAge = function() {
        return privateAge;
    }
}
let person = new Person("John", 30);
console.log(person.name); // "John"
console.log(person.privateAge); // undefined
console.log(person.getAge()); // 30
```

In this example, the privateAge variable is defined inside the constructor function and can only be accessed using the getAge method. This means that the internal state of the object is hidden from other objects and code, providing a level of protection and security.
Access modifiers such as private, public and protected are not natively supported in javascript, but it can be implemented using closures as well. In addition, other libraries and frameworks such as TypeScript provide a way to use access modifiers.
In conclusion, encapsulation is a fundamental concept in OOP that allows developers to hide the internal details of an object from other objects and code. In JavaScript, encapsulation is achieved by using closures and access modifiers, such as the private and public keywords. This allows developers to protect the internal state of an object and create more robust and maintainable code.
Encapsulation is not natively supported in javascript but it can be implemented using closures and other libraries like TypeScript.

INHERITANCE

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows developers to create a new class that inherits properties and methods from an existing class. In JavaScript, inheritance is achieved by using the prototype property and the Object.create() method.
JavaScript uses a prototype-based inheritance model, which means that objects can inherit properties and methods from other objects. Each object has a prototype property that refers to another object, and properties and methods are inherited by traversing the prototype chain.
For example, consider the following code:

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}

Person.prototype.greet = function() {
    return `Hello, my name is ${this.name} and I am ${this.age} years old.`;
}

function Student(name, age, major) {
    Person.call(this, name, age);
    this.major = major;
}

Student.prototype = Object.create(Person.prototype);
Student.prototype.constructor = Student;

let student = new Student("John", 25, "Computer Science");
console.log(student.greet()); // "Hello, my name is John and I am 25 years old."
console.log(student.major); // "Computer Science"
```

In this example, the Student class inherits properties and methods from the Person class. The Object.create() method is used to create a new object that inherits from the Person.prototype object, and the constructor property is reset to the Student class.
In JavaScript, the inheritance model is a bit different from other languages like Java or C#, where classes inherit from other classes and the subclass has access to the methods and properties of the superclass. In javascript, it's based on the prototype chain, where each object has a prototype property that refers to another object and it inherits properties and methods by traversing the prototype chain.
In conclusion, inheritance is a fundamental concept in OOP that allows developers to create a new class that inherits properties and methods from an existing class. In JavaScript, inheritance is achieved by using the prototype property and the Object.create() method. This allows developers to create reusable code and create a more organized and maintainable codebase. Understanding the prototype-based inheritance model in javascript is essential to be able to use inheritance effectively in javascript.

POLYMORPHISM

Polymorphism is a fundamental concept in object-oriented programming (OOP) that refers to the ability of different objects to respond to the same method call in different ways. In JavaScript, polymorphism is achieved by using function overloading and function overriding.
Function overloading refers to the ability of a function to have multiple implementations based on the number and/or types of arguments passed to it. JavaScript does not support function overloading natively, however, you can achieve similar functionality by using the arguments object and checking the number and/or types of arguments passed to the function.

Function overriding refers to the ability of a subclass to provide a different implementation of a method that is already provided by its superclass. In JavaScript, this can be achieved by reassigning the prototype of the subclass.
For example, consider the following code:

```
class Shape {
constructor(name) {
this.name = name;}draw() {
    console.log(`Drawing a ${this.name}`);}}class Circle extends Shape {draw() {
    console.log(`Drawing a Circle`);}}class Square extends Shape {draw() {
    console.log(`Drawing a Square`);}}let shape = new Shape("Shape");let circle = new
Circle();let square = new Square();

shape.draw(); //Drawing a Shape
circle.draw(); //Drawing a Circle
square.draw(); //Drawing a Square
```

In this example, the draw() method is overridden in the subclasses Circle and Square, providing a different implementation of the method that is already provided by the superclass Shape.
In conclusion, polymorphism is a fundamental concept in OOP that refers to the ability of different objects to respond to the same method call in different ways.

ABSTRACTION

Abstraction is a fundamental concept in object-oriented programming (OOP) that refers to the practice of hiding the implementation details of an object and exposing only the essential features to the user. In JavaScript, abstraction is achieved by using abstract classes and interfaces.

An abstract class is a class that cannot be instantiated and is meant to be used as a base class for other classes. Abstract classes typically contain one or more abstract methods, which are methods that have a signature but no implementation. These methods must be implemented by the derived classes.

JavaScript does not support abstract classes natively, but you can achieve similar functionality by using a combination of function constructors and prototypes.

For example, consider the following code:

```
function Shape() {
    if (this.constructor === Shape) {
        throw new Error("Cannot instantiate abstract class Shape");
    }
    this.draw = function() {
        throw new Error("Cannot call abstract method draw from Shape");
    }
}
```

```
function Circle() {
 Shape.call(this);
 this.draw = function() {
 console.log("Drawing a Circle");
 }
}
Circle.prototype = Object.create(Shape.prototype);
Circle.prototype.constructor = Circle;

let circle = new Circle();
circle.draw(); // "Drawing a Circle"
let shape = new Shape(); // Error: Cannot instantiate abstract class Shape
```

Abstraction is a fundamental concept in object-oriented programming (OOP) that refers to the practice of hiding the implementation details of an object and exposing only the essential features to the user. In JavaScript, abstraction is achieved by using abstract classes and interfaces. An abstract class is a class that cannot be instantiated and is meant to be used as a base class for other classes. Abstract classes typically contain one or more abstract methods, which are methods that have a signature but no implementation. These methods must be implemented by the derived classes. JavaScript does not support abstract classes natively, but you can achieve similar functionality by using a combination of function constructors and prototypes. For example, consider the following code: