I-Novotek Academy

Full Stack Web Development Course PDF Guide

website: www.inovotekacademy.com

Youtube: i-novotek academy

Course link: https://www.udemy.com/course/fullstack-web-development-course-projects-base/?referralCode=F8C808368D020D5794BD

The Basics of Javascript: A Beginner's Guide

If you're new to programming, javascript is a great language to start with. It's easy to learn and there are many resources available to help you get started. In section, we'll cover the basics of javascript, including variables, data types, operators, and functions. By the end of this section, you'll have a better understanding of how javascript works and be able to write basic programs.

What is Javascript?

Javascript is a programming language that allows you to add dynamic content to your web pages. That means you can create things like animations, games, and form validation with javascript. It runs on your web browser and doesn't require a separate download or installation like some other languages.

`

Javascript is what's called a "client-side" language. That means the code runs on your computer, not on the server where the website is hosted. This is in contrast to "server-side" languages like PHP or ASP, which run on the server before the page is even sent to your browser.

Client-side languages are convenient because they don't require any special setup on the server. All you need is a text editor and a web browser, and you're ready to start coding!

How Does Javascript Work?

When you view a web page, your browser downloads the HTML code from the server and displays it as a webpage. Javascript code is embedded into the HTML code and runs automatically when the page loads.

The Benefits of Javascript

Javascript is a programming language that is widely used by web developers to create interactive web applications.

Javascript has a number of benefits that make it a popular choice for web development. It is easy to learn, versatile, and can be used to create a wide variety of applications. Additionally, javascript is free to use and is supported by all major web browsers.

If you are considering learning a programming language for web development, javascript is a great choice. It is a versatile language that can be used to create a wide variety of applications.

With Javascript you can become a frontend, backend and fullstack developer

With Javascript you can create web applications, mobile and desktop apps

Javascript Variables: A Comprehensive Guide

What are variables in JavaScript?

JavaScript variables are used to store data values. In JavaScript, data values can be text strings, numbers, arrays, or objects. Variables can be declared using the var, let, or const keywords.

How to create variables in JavaScript

We use the following keywords to create variables

- **var**

- **let**
- **const**

var keyword

the var keyword is used to create a variable that is globally scoped, meaning it can be accessed from anywhere in your code.

let keyword

The let keyword is used to create a variable that is locally scoped, meaning it can only be accessed from within the block of code it was declared in.

const keyword

The const keyword is used to create a variable that cannot be reassigned, making it a constant value.

To declare a variable, you will use the following syntax:

var variableName = value;

Where variableName is the name of your variable and value is the value you are assigning to the variable. You can also declare

multiple variables on the same line using a comma delimiter like this:

```
var name = "John Doe", age = 39, isMarried = true;
```

Once a variable is declared, you can assign it a value using the equal sign (=):

myName = "John Smith"; // Assigns the value "John Smith" to the myName variable.

You can also declare multiple variables without assigning values to them by leaving off the value part of the statement. Variables declared in this way will have a default value of undefined.

var name, age, isMarried; // all variables are assigned the value of undefined by default. /* The following shows data types */

Basic rules of JavaScript syntax

Here are some of the basic rules that govern JavaScript syntax.

1. JavaScript is case-sensitive. This means that language keywords, variables, function names, and any other identifiers must always be

typed with a consistent capitalization of letters. The keyword var is not the same as the keyword var.

2. Statements must end with a semicolon (;). For example, the statement x = 5; is valid, but the statement x = 5 is not.

3. Whitespace (spaces, newlines, and tabs) is generally ignored, except when it is used to separate tokens. For example, the statements x=5 and x = 5 are equivalent.

4. Comments can be added to your code to make it more readable. JavaScript supports two types of comments:

// This is an single-line comment /* This is a multi-line comment */

5. Identifiers can be any combination of letters, digits, underscores (_), and dollar signs ($). However, they cannot start with a digit. In addition, some reserved words cannot be used as identifiers (e.g., class, return, etc.).

6. Variables must be declared before they are used. This can be done using the keyword var. For example:

var x; // declares a variable named x var y = 5; // declares a variable named y and assigns it the value 5

7. Data types in JavaScript include numbers, strings, Booleans (true/false values), and objects. There are also two special data types: undefined and null.

8. Numbers can be written with or without decimals. For example, 3.14, 42, and -99 are all valid numbers.

9. Strings must be enclosed in quotes. Single or double quotes can be used, but they must match (e.g., "hello" and 'goodbye' are both valid, but "hello' is not). Strings can span multiple lines by using the backslash (\ ) as an escape character:

"This is the first line.\nAnd this is the second." // produces "This is the first line.<newline>And this is the second."

10 .Code must be put within <script> tags in order to run

11 . Javascript is a text-based language. This means that it is made up of words, numbers, and punctuation marks that are read by a computer and interpreted into instructions.

Variable naming conventions in JavaScript

- **Variable names can start with letters**, $, or _ .

The following variable names are all valid:

```
$sname, sname, name_sname.
```

- **No spacing**

Variable names should not contain spaces use an underscore instead.

```
var user_name = "John Doe";
```

- **Use descriptive names:**

Using descriptive names for your variables is a good way to make your code more readable and understandable. When choosing a name for your variable, try to think of a word or phrase that accurately describes what the variable is used for.

- **Use camel case:**

Camel case is a naming convention where each word in the name is capitalized, except for the first word. For example, the variable name “myVariable” would be written in camel case.

Do not use reserved words:

When choosing a name for your variable, be sure to avoid using any of the reserved words in JavaScript. Reserved words are words that have special meaning in the language and cannot be used as variable names.

- **Do not use spaces or special characters:**

Spaces and special characters are not allowed in JavaScript variable names. If you need to use multiple words to describe your variable, you can use camel case orunderscores to separate the words (“my_variable”).

- **Keep it short:**

Long variable names can make your code more difficult to read and understand. When possible, try to choose a name that is short but still descriptive.

Javascript data types

The JavaScript language contains two different types of data: primitive values and objects.

A primitive value is a value that has no properties or methods. A primitive value is immutable, which means it cannot be changed. The only way to create a new primitive value is to create a new variable and assign it a new value.

There are the primitive values in JavaScript: undefined, null, Boolean, Number, String, Symbol, and bigint.

1. undefined

Undefined is a value that represents no value. It is the default value of variables that have not been assigned a value.

2. null

Null is a value that represents no value. It is used to indicate that a variable does not have a value.

3. Boolean

Boolean is a value that represents either true or false.

4. Number

A number is a value that represents a number. All numbers in JavaScript are 64-bit floating-point numbers.

5. String

String is a value that represents a sequence of characters. Strings are immutable, which means they cannot be changed. The only way to create a new string is to create a new variable and assign it a new value.

6. Symbol

Symbol is a unique and immutable value that can be used to identify an object. Symbols are typically used as keys in objects.

7. **Objects (collections of properties)**

Javascript Operators: A Comprehensive Guide

What are operators in JavaScript?

Operators in JavaScript are the symbols that represent certain actions that can be performed on variables. In JavaScript, there are many different kinds of operators, including arithmetic operators, assignment operators, comparison operators, logical operators, and Bitwise operators. Each kind of operator performs a different kind of operation.

Arithmetic operators

Arithmetic operators take two operands (values) and perform an arithmetic operation on them

Arithmetic operators include

- **Addition (+)** to add operands together example let sum = 2+6
- **Subtraction (-)** subtracts operands together let difference = 9-7
- Multiplication (*)
- **Division (/)** let total = 9/2
- **Modulus (%)** division with remainder. Example 12%2 would give you 0 as the remainder while 13%2 would give you 1 as it should be according to division;

Increment (++)

Synthax

- **x++ (** postfix**)** Postfix increment
- **++x (**prefix) Prefix increment

Postfix increment

If used postfix, the increment operator will first increment the value before returning it.

```
let x = 3;
y = x++;
// y = 3
// x = 4
```

Prefix increment

If used as a prefix operator (for example, ++x), the increment operator will increment the operand and return the new value.

```
let a = 2;
b = ++a;
// a = 3
// b = 3
```

Assignment operators

The assignment operator in javascript assigns a value to a variable. The most common assignment operator is the = operator, which assigns the value to the left of the = to the variable on the right. For example, if we have a variable called x and we want to give it the value of 5, we would write x = 5.

Other assignment operators include the += operator, which adds the value to the left of the = to the variable on the right, and the -= operator, which subtracts the value to the left of the = from the variable on the right. Example :

int x = 5; // Assigns 5 to x x -= 4; // Subtracts 4 from x, so now x equals 1 int y = 4; y += 2; // Adds 2 to y, so now y equals 6

Combined Assignment Operators

You can combine the assignment operator with basic arithmetic operators using the combined assignment operators (+= , -= , *= , /= , &= , and |=). This lets you write code like this:

let m=0

m = m + 5 is equivalent to m += 5 .

m = m * 3 is equivalent to m *= 3 .

Double equal to vs tripple equal to

Double equal to is == , it compares values.

Tripple euqal to is === , it compares both value and type.

Example of ==

"" == "0" is true   then when comparing string , == will compare only values, not type.

```
 0 == false is also true, here as you can see when comparing
value 0 and false then it converts 0 to falsy value.
```

Example of ===

```
= "" === "0" is false, because it compares both values and
their data types(string vs number)
```

Comparison operators

Comparison operators Comparison operators are used to compare two values:

- greater than (>)
- less than (<)
- greater than or equal to (>=)
- and less than or equal to (<=).

NOTE: These operators evaluate expressions such that they return either true or false :

3 < 5 // evaluates to true 3 > 5 // evaluates to false 3 >= 4 // evaluates to true 2 <= 1 // evaluates

Logical operators

Logical operators allow JavaScript to perform boolean logic on values. These operators are:

- **and’ ,bb’** : If each of the two operands is true, then the condition becomes true. (a && b) will be true only if both a and b are true.

- **or’ ‡ÇÂ•** : If any of the two operands is true, then the condition becomes true. (a || b) will be true if either a or b is true.

- **not’ , “** Used with boolean values to reverse the value i.e. from false to true, and from true to false

  var x = 2;

  // x != 7 is TRUE

Conditional Statements in javascript

What is a conditional statement? A conditional statement is a set of commands that will only be executed if a specified condition is met.

```
if (condition) {

  // commands to execute if condition is true

} else {

  // commands to execute if condition is false

}
```

If, else if, and else

are all used to create conditional statements in various programming languages.

Nested conditional statements A nested conditional statement is a conditional statement that contains another conditional statement as one of its components. Code example

Nested conditional statements

Truthy and falsy values in javascript

In JavaScript, a truthy value is a value that is considered true when evaluated in a Boolean context. All values are truthy unless they are defined as falsy. The following values are considered falsy:

- false

- 0

- ""

- null

- undefined

So what does this all mean? Basically, any value that is not one of the falsy values listed above is considered truthy. This includes values like true, 1, and "foo".

JavaScript Loops

JavaScript loops are a powerful programming tool that allows you to execute a block of code multiple times. While there are many different types of loops, they all share a common structure: a condition is checked, and if the condition is true, the code block is executed. This process is then repeated until the condition is no longer true. In this blog post, we'll take a closer look at how JavaScript loops work and how you can use them in your own pr

ograms.

The For Loop

The for loop has the following syntax: > The **for** statement creates a loop that consists of three optional expressions, enclosed in parentheses and separated by semicolons, followed by a statement or a block of statements.

```
for ([initialization]; [condition]; [final-expression]){ //
code here... }
```

```
inal-expression ===incrementer/decrementer
```

> The **initialization** expression initializes the loop; it's executed once, as the loop begins.

> When the **condition** returns true, the loop executes the statement.

> The **final-expression** is executed every time the statement (s) runs.

Code example :

```
let i;

for(i=0;i<5; i++){ // code here will run 5 times! }
```

The while loop

The while loop is similar to the for loop, but instead of using an initializer and an incrementer, it only has a condition. The code block is executed repeatedly until the condition evaluates to false.

```
 while (condition) {

  code block to be executed}

  }
```

let i = 0;

```
  while (i < 10) {

    console.log(i);

    i++;

 } // loops ...0, 1, 2,...9
```

The While Loop

The while loop loops through a block of code as long as a specified condition is true.

```
while (condition) {
  // code  to run
}

let msg = "";
while (x < 10) {
 msg += "The number is " + i;
 i++;
}
```

Javascript Functions

What is a function?

In JavaScript, a function is a piece of code that is written to perform a specific task. Functions are usually self-contained and can be reused across your code. This makes them very important in JavaScript programming.

Why functions

There are many reasons why you should use functions in JavaScript. Here are 10 of the most important reasons:

1. Functions help to make your code more readable.

2. Functions can make your code more reusable.

3. Functions can help to make your code more maintainable.

4. Functions can improve the performance of your code.

5. Functions can help to modularize your code.

6. Functions can make your code more testable.

7. Functions can help to reduce the complexity of your code.

8. Functions can improve the flexibility of your code.

9. Functions can improve the

Defining functions

Functions can be written either as a

- function declaration or a
- function expression.

Function declarations are written as follows:

```
function name() {
  // code to be executed
}
```

Function expressions are written as follows:

```
let name = function() {
  // code to be executed
}
```

Function return keyword

The return keyword is used to exit a function and return a value to the caller.

The return keyword can be used with or without a value. If a value is present, it is returned to the caller. If no value is present, the function simply exits.

The return keyword is essential for creating functions that return values. Without it, functions would only be able to perform actions, not return values to the caller.

This would limit the usefulness of functions and make them much less powerful.

So if you're creating a function that needs to return a value, be sure to use the return keyword. It'll make your function much more useful and powerful.

Differences between function arguement and function parameters

Function arguments and function parameters are often used interchangeably, but there is actually a subtle difference between the two. Function arguments are the values that are passed to a function when it is invoked, while function parameters are the variables that are used to receive those arguments.

Javascript Strings: The Essential Guide

Introduction

Javascript strings are one of the most essential data types in the language. In this guide, we will explore the various properties and methods associated with strings, as well as how to manipulate and format string data.

What are strings?

In Javascript, strings are lines of text that are used to store and represent data. Strings can be anything from a single character to an entire novel—any sequence of characters can be stored as a string. Strings must always be placed within either single or double quotation marks (' ' or " ").

Strings are commonly used for storing user input, such as when a user enters their name into an input field on a website.

How to create strings?

Strings can be created in several different ways. The most common way is simply by assigning a string of characters to a variable:

```
const myName = 'Paige' ;

const greeting = 'Hi there!' ;

 const favoriteNumber = 'My favorite number is 7'

 let str1 = 'Hello' , str2= ', world!' ;
```

toLowerCase()

toUpperCase()

String length()

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

The length property of an empty string is 0.

```
let str = "Hello World!";

let length = str.length;
```

String trim()

The trim() method is useful for removing whitespace from both sides of a string. This can be helpful for cleaning up user input, or for making sure that two strings are truly equal.

The trim() method does not mutate the original string.

```
let str = "     Coding time        ";

let result = str.trim();
```

String split()

The split() method separates a string into an array of substrings.

The split() method returns a new array.

The split() method does not change the original string.

```
let str3 = "Are you coding today?";

const myArray = str3.split("");
```

string reverse()

The reverse() method changes the order of the elements in an array so that the first element becomes the last, the second element becomes the second to last, and so on.

The reverse() method modifies the original array.

```
const food = ["Pizza", "Congee", "Fufu", "rice"];

const newFood = food.reverse();
```

Array join()

The join() method returns an array converted to a string.

The join() method does not change the original array.

Any separator can be specified. The default is comma (,).

```
const languages = ["English", "Twi", "French", "Fante"];

let text = languages.join();

console.log(text);
```

String repeat()

The slice() method produces a new string that is a copy of the original string. The original string is not changed. === making copies of strings

```
let str = "Hello world!";

let result = str.repeat(3);
```

String startsWith()

The startsWith() method is useful for determining whether a string begins with a specified string.

This is important because it can help ensure that a string is the correct format.

For example, if you are expecting a string to be in all caps, you can check to see if the string starts with a capital letter.

If the string does not start with a capital letter, you know that it is not in the correct format.

The startsWith() method is case sensitive.

```
let str = "Welcome to i-novotek Academy";
str.startsWith("Welcome");
```

String includes()

- The includes() method determines whether a string contains a specified string.

  If it does, it returns true.Otherwise, it returns false.
- It takes two arguments ('text to search', 'startting point') //default is 0
- The includes() method is case-sensitive.

```
let str = 'Are you a web developer';
```

let result = str.includes('you');

```
let result2 = str.includes('you', 4);
```

concat() Method

The concat() method concatenates two or more strings.

The concat() method does not mutate the original strings.

The concat() method returns a new string.

```
let str1 = 'Your';

let str2 = 'order';

let str3 = 'is ready';

let result = str1.concat(' ', str2, ' ', str3);
```

slice() Method

The slice() method extracts a section of a string.

The initial position is 0, the subsequent is 1,

A negative value picks from the end of the string.

The slice() method does not alter the original string.

The slice() method enables you to select from a given start point up to (but not including) a given end point. This method does not change the original data.

```
let str = 'Welcome to javascript methods';

let result = str.slice(0, 5);
```

let ans = str.slice(3);

```
console.log(ans);
```

String comparison

In order to compare whether one string is greater than another, JavaScript employs what is known as "dictionary" or "lexicographical" order.

In other words, strings are compared letter-by-letter.

```
console.log( 'Z' > 'A' ); // true

console.log( 'Glow' > 'Glee' ); // true

console.log( 'Bee' > 'Be' ); // true
```

FACTS ABOUT STRING COMPARISMS

- Compare the first character of both strings
- First it will check the first letters on both sides if one is greater than the other it will return true but if they are equal it will move to the next letter until it returns true /false
- Lowercase letters are greater than uppercase letters of the same type Because the lowercase character has a greater index in the internal encoding table JavaScript uses (Unicode)
- If both strings end at the same length, then they are equal. Otherwise, the longer string is greater

Comparison of different types

- When comparing values of different types, JavaScript converts the values to numbers.

```
console.log( '3' > 2 ); // true, string '3' becomes a number 3

console.log( '05' == 3 ); // true, string '05' becomes a number
5
```

- For boolean values, true becomes 1 and false becomes 0

```
console.log( true == 1 ); // true

console.log( false == 0 ); // true
```

Comparison with null and undefined

For a strict equality check ===
These values are different, because each of them is a different type.

```
console.log( null === undefined ); // false
```

For a non-strict check ==
There’s a special rule. These two are a “sweet couple”: they equal each other (in the sense of ==), but not any other value.

```
console.log( null == undefined ); // true
```

JAVSSCRIPT ARRAYS

arrays are a collection of items in a particular order and can be accessed by index number.

create an array

Method 1: using the new keyword
Example:

```
const myArray1 = new Array("css", "html", "javascript");

//or

const myArray2 = new Array();

myArray2[0] = "pizza";

myArray2[1] = "burger";

myArray2[2] = "chicken";
```

Method 3: using the array literal

```
const myArray3 = ["Node", "Express", "MongoDB"];

//or

const myArray4 = [];
```

```
myArray2[0] = "Node";
myArray2[1] = "Express";
myArray2[2] = "MongoDB";
```

Accessing elements in an array

```
const myArray5 = ["Node", "Express", "MongoDB"];
console.log(myArray5[0]);
```

Iterating over an array

method 1: for loop

```
for (let i = 0; i < myArray5.length; i++) {
  console.log(myArray5[i]);
}
```

Advanced: method 2: forEach //We will discus later

```
myArray5.forEach(function (element) {
  console.log(element);
});
```

Array methods

Method 1: push()

adds an element to the end of an array

```
const myArray6 = ["Node", "Express", "MongoDB"];
myArray6.push("React");
console.log(myArray6);
```

Method 2: pop()

removes an element from the end of an array

```
const myArray7 = ["Node", "Express", "MongoDB"];
myArray7.pop();
console.log(myArray7);
```

Method 3: unshift()

adds an element to the beginning of an array

```
const myArray8 = ["Node", "Express", "MongoDB"];
myArray8.unshift("React");
console.log(myArray8);
```

Method 4: shift()

removes an element from the beginning of an array

```
const myArray9 = ["Node", "Express", "MongoDB"];
myArray9.shift();
console.log(myArray9);
```

Method 5: indexOf()

returns the index of the first element in an array that matches the specified value

```
const myArray10 = ["Node", "Express", "MongoDB"];
console.log(myArray10.indexOf("Express"));
```

Method 6: lastIndexOf()

returns the index of the last element in an array that matches the specified value

```
const myArray11 = ["Node", "Express", "MongoDB"];

console.log(myArray11.lastIndexOf("Express"));
```

Method 7: includes()

returns a boolean indicating whether an array includes a certain value

```
const myArray12 = ["Node", "Express", "MongoDB"];

console.log(myArray12.includes("Express"));
```

Javascript objects

What are objects?

Objects are variables that hold multiple pieces of information. In JavaScript, objects are created with curly braces {}.

Creating Objects

method 1: Using the Object() constructor function:

```
const person = new Object();

person.name = "John";

person.age = 30;

person.city = "New York";
```

method 2: Using the object literal syntax:

```
const person2 = {
  name: "John",
  age: 30,
  city: "New York",
};
```

Accessing Object Properties

1. using dot notation:

```
person.name;
person.age;
person.city;
```

2. using bracket notation:

```
person["name"];
person["age"];
person["city"];
```

Updating Object Properties

1. using dot notation:

```
person.name = "Jane";
person.age = 31;
person.city = "Miami";
```

2. using bracket notation:

```
person["name"] = "Jane";
person["age"] = 31;
person["city"] = "Miami";
```

Deleting Object Properties

1. using dot notation:

```
delete person.name;
delete person.age;
delete person.city;
```

2. using bracket notation:

```
delete person["name"];
delete person["age"];
delete person["city"];
```

Adding Methods to Objects

```
const carObj = {
  make: "Ford",
  model: "Mustang",
  year: 1969,
  color: "red",
  description: function () {
    return `${this.make} ${this.model} (${this.year})`;
  },
};
```

What is this?

- In JavaScript, the this keyword refers to an object.
- The this keyword refers to different objects depending on how it is used:
- In an object method, this refers to the object.
- Alone, this refers to the global object.
- In a function, this refers to the global object.
- In a function, in strict mode, this is undefined.
- In an event, this refers to the element that received the event.

Iterating Over Objects

1. using for...in loop:
2. using Object.keys() method:
3. using Object.values() method:
4. using Object.entries() method:

1. for...in loop:

Syntax:

```
for (let key in obj) {

 //code to run

}

.key is the name of the property

.in is the keyword
```

.obj is the object

```
//Example:

const student = {

  name: "John",

  age: 30,

  city: "New York",

};

//iterating over an object using for...in loop
```

//-----

```
for (let key in student) {

  console.log(key);

}
```

2. using Object.keys() method:

Syntax

```
Object.keys(obj)
```

obj is the object

```
//Example:

const student2 = {
  name: "John",
  age: 30,
  city: "New York",
};
//iterating over an object using Object.keys() method
//-----
//convert the keys of an object into an array
const keys = Object.keys(student2);
//console.log(keys); //array
```

iterate using forEach loop

Syntax:

```
//obj.forEach(function(value, key) {
//  //code to run
//}
//value is the value of the property
//key is the name of the property
keys.forEach((key) => {
  console.log(key);
  console.log(`${key}: ${student2[key]}`);
});
//3. using Object.values() method:
```

Object.values()

/The Object.values() method works opposite to that of Object.key(). It returns the values of all properties in the object as an array. You can then loop through the values array by using any of the array looping methods.

```
//Syntax:

//Object.values(obj)

//obj is the object

//-----
```

Example:

```
const student3 = {

  name: "John",

  age: 30,

  city: "New York",

  height: 1.8,

  weight: 80,

};
```

iterating over an object using Object.values() method

```
//convert the values of an object into an array

const values = Object.values(student3);

console.log(values);
```

looping through an array of objects using forEach loop

```
values.forEach((value) => {

  console.log(value);
});
```

4. Using object.entries() method:

The object.entries() method returns an array of arrays, where each array contains a key and value pair.

The object.entries() method is useful when you want to iterate over the properties of an object.

Example

```
const student4 = {
  name: "John",
  age: 30,
  city: "New York",
  height: 1.8,
  weight: 80,
};
```

iterating over an object using Object.entries() method

//convert the entries of an object into an array

```
const entries = Object.entries(student4);
console.log(entries);
```

using forEach loop

```
entries.forEach((entry) => {
  console.log(entry);
});
```

destructuring assignment

```
entries.forEach(([key, value]) => {
  console.log(`${key}: ${value}`);
});
```

Javascript versions

JavaScript and its current state.

JavaScript was created by Brendan Eich in 1995 and became an ECMA standard in 1997. The official name of the language is ECMAScript. ECMAScript versions have been abbreviated to ES1, ES2, ES3, ES5, and ES6. Since 2016, new versions have been named by year (ECMAScript 2016 / 2017 / 2018).

1. Introduction
2. What's new in ECMAScript 6?
3. What's new in ECMAScript 7?
4. What's new in ECMAScript 8?
5. What's new in ECMAScript 9?
6. What's new in ECMAScript 10?
7. Conclusion

The complete guide to understanding numbers in Javascript

Number Basics In JavaScript, numbers are a primitive data type. This means that they are not an object and they have no methods.

Numbers can be positive or negative, integers or floats. An integer is a whole number, while a float is a number with a decimal point.

Example fo floating numbers

```
const num = 0.1 + 0.2;

console.log(num);
```

Example fo integer numbers

```
const num = 8;

console.log(num);
```

Infinity (or -Infinity)

Infinity (or -Infinity) is the value JavaScript will return if you calculate a number outside the largest possible number.

```
const infinity = 1 / 0;

const infinity2 = 1 / Infinity;

const infinity3 = -1 / 0;
```

Numerical operations

```
const number1 = 10;

const number2 = 2;

const number3 = number1 + number2;
```

Numerical string operations

Javascript converts string to number in numerical operations

```
const number4 = "10";

const number5 = 2;

const number6 = number4 + number5;
```

Numerical string operations with strings

This won't work because Javascript +operator to concatenate strings

```
const number7 = "10";

const number8 = "2";

const number9 = number7 + number8;
```

NaN - Not a Number (Not legal number)

Example of NaN

```
const number10 = "10";

const number11 = 2;

const number12 = number10 / number11;
```

Example2

```
const number13 = "10";
const number14 = "foo";

const number15 = number13 / number14;
```

JavaScript Number Methods

What is a number in javascript

A number in JavaScript is a double-precision 64-bit number that can represent anything from an integer to a fraction to a very large or very small real number.

Primitive values cannot have properties and methods, but JavaScript treats primitive values as objects when executing methods and properties. This allows for methods and properties to be available to primitive values.

toFixed()

The toFixed() method rounds a number to a certain number of decimals and returns a string.

```
const num1 = 10.3;

const num2 = 10.6;

const num3 = num1.toFixed(2);

const num4 = num2.toFixed(2);
```

toString()

The toString() method converts a number to a string.

```
const num5 = 10;

const num6 = num5.toString();
```

Converting various types to numbers

parseInt()

- The parseInt() method parses a string and returns an integer.

- Only first number is returned by ignoring the rest of the string

```
const num7 = "10.909";

const num8 = parseInt(num7);
```

parseFloat()

- The parseFloat() method parses a string and returns a floating point number.
- It returns all numbers including the decimal point.

```
const num9 = "10.3";

const num10 = parseFloat(num9);
```

Number()

The Number() method converts a string to a number.

```
const num11 = "10";

const num12 = Number(num11);
```

1. What is the result of calling toString() on a number?

2.What is the result of calling toFixed() on a number?

3.How would you round a number up or down?

4.How do you check if a number is an integer?

5.What is the result of calling valueOf() on a number?

6.How would you find the highest or lowest value in an array of numbers?

7.How do you check if a number is positive or negative?

8.How would you calculate the absolute value of a number?

9.What is the result of calling toExponential() on a number?

10.What is the result of calling toPrecision() on a number? 11.What is the result of calling toLocaleString() on a number?

12.How would you check if a number is NaN?

13.How would you check if a number is infinite?

14.How would you find the modulus of two numbers?

15.What is the result of calling toJSON() on a number?

16.What base does JavaScript use for numeric literals?

17.Only valid numeric strings can be converted into numbers, what value is produced when an invalid string is converted?

18.Is it possible to perform arithmetic operations on non-numeric values? If so, how do they work?

19.What value does 0 in x10 represent in JavaScript? How about 0 in x16?

20.In which cases does a – b produce a different result from b – a ?

JAVASCRIPT MATH OBJECT

Everything You Need To Know About The JavaScript Math Object

What is the Math Object?

The Math object is a built-in object that has properties and methods for mathematical constants and functions. The Math object is automatically created by the JavaScript engine and is available in all browsers.

Comparing Numbers There are three ways to compare numbers:

//

The == operator The === operator The != (= not equal) and !== (not equal value or not equal type) operators

Math.abs(x)

Returns the absolute value of x. (Thus only positive numbers will be returned,)

```
//Example: Math.abs(-5) returns 5.

//Example: Math.abs(5) returns 5.
```

Math.round(x)

Returns the value of x rounded to the nearest integer.

```
//Example: Math.round(5.5) returns 6.

//Example: Math.round(5.4) returns 5.
```

Math.ceil(x)

//Returns the value of x rounded up to the nearest integer.

```
//Example: Math.ceil(5.4) returns 6.

//Example: Math.ceil(5.5) returns 6.
```

Math.floor(x)

Returns the value of x rounded down to the nearest integer.

```
//Example: Math.floor(5.4) returns 5.
```

```
//Example: Math.floor(5.5) returns 5.
```

Math.sqrt(x)

Returns the square root of x.

```
//Example: Math.sqrt(25) returns 5.
```

Math.pow(x, y)

Returns the value of x to the power of y.

```
//Example: Math.pow(2, 3) returns 8.
```

Math.min(x, y)

Returns the smaller of x and y.

```
//Example: Math.min(5, 10) returns 5.

//Example: Math.min(10, 5) returns 5.
```

Math.max(x, y)

Returns the larger of x and y.

```
//Example: Math.max(5, 10) returns 10.

//Example: Math.max(10, 5) returns 10.
```

Math.random()

Returns a random number between 0 and 1.

```
//Example: Math.random() returns a number between 0 and 1.

//Example:  Math.floor(Math.random() * 10) returns a number
between 0 and 9.
```

Get random number between two numbers

//syntax

```
 Math.floor(Math.random() * (max - min + 1)) + min
```

function to get random number between two numbers

```
function getRandomNumber(min, max) {

  return Math.floor(Math.random() * (max - min + 1) + min);

}
```

Javascript advance (ES6)

Arrow functions

An arrow function expression is a more concise way to write a traditional function expression, but it cannot be used in all situations.

Characteristics of Arrow functions

1. Arrow functions do not have this keyword.

```
const carObj = {

  name: "BMW",

  getName: () => {

    return this.name;

  },

};

// carObj.getName(); //null
```

2. Arrow functions do not have arguments.

```
const add = () => {
  console.log(arguments);
};
```

4. Arrow functions cannot use as a constructor.

```
const Person = name => {
  this.name = name;
};
 const p = new Person("John");
```

function constructor

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

Arrow functions syntax

1.With one parameter with expression

return keyword and parenthesis are not necessary

```
const add2 = b => b;
add2(2); //3
```

2. multiple parameters with expression

requires parentheses , return keyword is not necessary

```
//(parameter1, parameter2, ...) => expression

const add3 = (a, b) => a + b;
```

3. Multiline statements

require body braces and return

```
const user = a => {

  let b = 10;

  return a + b;

};
```

4. Multiple parameters with multiple statements

```
const add4 = (a, b) => {

  let c = 10;

  let d = 20;

  return a + b + c + d;

};
```

5. Imediately return object literal

```
const userInfo = user => ({ name: user.name, age: user.age });
```

6. Destructuring object with arrow functions

```
const user2 = {

  name: "John",
  age: 30,
};
const displayUser = ({ name, age }) => {
  console.log(name);
  console.log(age);
};
displayUser(user2);
```

7. Destructuring array with arrow functions

```
 const displayData = ([name, age]) => {
  console.log(name);
   console.log(age);
 };
 displayData(["John", 30]);
```

```
const displayData = ([name, age] = ["John", 50]) =>
  `Your name is ${name} and your age is ${age}`;
displayData();
```

8. Arrow functions with rest parameters

(parameter1, parameter2, ...) => expression

```
const findMin = (...numbers) => Math.min(...numbers);
```

Coding challenge 1

```
const add5 = (a, b, ...rest) => {
  console.log(rest);
  return a + b + rest.reduce((acc, cur) => acc + cur);
};
add5(1, 2, 3, 4, 5); //15
```

9. Arrow functions with default parameters

(parameter1 = value, parameter2 = value, ...) => expression

const add8 = (a = 10, b) => a + b;

```
const add6 = (a = 10, b = 20, ...rest) =>
  a + b + rest.reduce((acc, cur) => acc + cur);
```

Spread Operator

The spread operator (...) is a convenient way to copy all or part of an existing array or object into another array or object.

Difference between Spread (...) and Rest (...) operator

Spread syntax "expands" an array into its elements, while rest syntax collects multiple elements and "condenses" them into a single element

Spread with strings

```
console.log(..."hello"); //h e l l o
```

Spread with array

```
console.log([..."nodejs"]); //[1, 2, 3]

const arr = [1, 2, 3];

const arr2 = [4, 5, 6];
```

combine the two arrays

method 1: using concat

```
const newArr = arr.concat(arr2);

//console.log(newArr); //[1, 2, 3, 4, 5, 6]
```

method 2: using spread operator

```
const newArr2 = [...arr, ...arr2];

console.log(newArr2); //[1, 2, 3, 4, 5, 6]
```

Copy an array

```
const arr = [1, 2, 3];
```

const arr2 = [...arr];

```
arr2.push(4);
```

Spread with objects

The spread operator can be used to spread an object into a list of arguments.

Example 1

```
const obj = {

  name: "John",
  age: 30,

};


const obj2 = {

  name: "Mary",

  age: 25,

};


const obj3 = {

  city: "New York",

  country: "USA",

};
```

method 1: using Object.assign

```
const newObj = Object.assign({}, obj, obj2, obj3);
```

NOTE: if two objects have the same property, the value of the second object will overwrite the value of the first object. so the last object will be the one that will be used.

```
console.log(newObj); //{name: "Mary", age: 25}
```

method 2: using spread operator

```
const newObj2 = { ...obj, ...obj2, ...obj3 };

console.log(newObj2); //{name: "Mary", age: 25, city: "New
York", country: "USA"}
```

CODE EXAMPLE

```
const qty = [1, 2, 3];

const prices = [10, 20, 30];
```

find the maximum price

```
const maxPrice = Math.max(...prices);

const maxQty = Math.max(...qty);
```

minimum price

```
const minPrice = Math.min(...prices);

const minQty = Math.min(...qty);
```

find the sum of maximum values of qty and prices

```
const max = Math.max(...qty, ...prices);

//console.log(max); //30
```

Function scope in javascript

How does function scope work in JavaScript?

Function scope in JavaScript refers to the visibility of variables within a function. Variables declared within a function are only accessible within that function. They are not accessible outside of the function.

In JavaScript, variables can belong to either the local or global scope.

The local scope is created when a function starts executing, and the global scope is created when your JavaScript code is loaded into the browser. If you create a variable without using the var keyword, it will be automatically added to the global scope. Like other programming languages, JavaScript also has something called lexical scoping. This means that nested functions have access to the outer function’s variables. In other words, functions can access outer function’s variables but not vice versa

global scope in javascript

In JavaScript, global scope refers to the root scope of the program. Global scope is the default scope for variables and functions defined outside of any other scope. This means that variables and functions defined in the global scope are accessible from anywhere in the program.

There are two ways to create variables and functions in the global scope.

1. The first is to simply define them outside of any other scope.
2. The second is to use the global keyword.

Variables and functions defined in the global scope are accessible from anywhere in the program. This can be useful for creating utility functions or variables that need to be accessed from different parts of the program. However, it can also lead to problems if variables or functions in the global scope are accidentally overwritten or modified.

It is generally best practice to avoid using global scope unless necessary.

Global scope variables

```
const user = {

  name: "John",

  age: 30,

};

const amountOfMoney = 100;

const sayHello = function () {

  return "Hello";

};
```

Create a function to access the variables and functions in the global scope

```
function getUserInfo(user) {

  return `${user.name} is ${user.age} years old

  `;

}

console.log(getUserInfo(user));
```

Create a function to mutate the variables and functions in the global scope

```
function changeUserInfo(user) {
  user.name = "Jane";
  user.age = 25;
}
console.log(getUserInfo(user));
console.log(user);
```

Block scope in javascript

In JavaScript, block scope refers to the visibility of variables within a code block. A code block is a set of curly braces {} that encloses one or more statements. Variables declared within a code block are only visible within that block and are not accessible from outside the block.

One of the most common uses of block scope is with the if statement. The code block associated with an if statement is only executed if the condition evaluates to true. This means that variables declared within the code block will only be accessible within the if statement.

Block scope can also be used with functions. Variables declared within a function are only visible within that function and are not accessible from outside the function.

Overall, block scope is a way of limiting the visibility of variables to only the code block in which they are declared. This can be helpful in preventing namespace collisions and making code more readable.

Exanple 1: using if statement

```
let age = 30;
if (age > 18) {
  let message = "You are old enough";
  console.log(message);

}
```

Exanple 2 using loop

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

  const name = "John";

  //console.log(i);

}
```

Function scope in javascript

In JavaScript, function scope refers to the visibility of variables within a function.

Variables that are declared within a function are only accessible within that function. This means that if you try to access a variable that is declared inside a function from outside the function, you will get an error.

Function scope is important to understand because it can impact the way your code runs. For example, if you have a variable that is declared inside a function, you will not be able to access that variable from outside the function. This can be helpful if you only want a certain piece of code to run under certain conditions. However, it can also lead to errors if you're not careful.

When you're working with function scope, it's important to keep track of your variables and where they are declared. This will help you avoid errors and make sure your code

```
function myFunction() {

  console.log(x);

  let x = "Hello";

}

myFunction();

console.log(x);
```

Lexical scope in javascript

In javascript, lexical scope refers to the visibility of variables and functions in relation to the lexical structure of the code. That is, the scope of a variable or function is determined by its position in the code. For example, variables and functions declared inside a function are only visible inside that function.

Lexical scope is an important concept in javascript because it affects how code is executed and how variables are accessed. It is also a complex topic, and there are a few different ways to think about it. In this article, we will explore lexical scope in javascript and how it works.

One way to think about lexical scope is in terms of the nesting of functions. When a function is declared inside another function, it is said to be lexically nested inside that function. The scope of the inner function is said to be lexically enclosed by the scope of the outer function. This means that the inner function has access to the variables and functions declared in the outer function, but not vice versa.

```
function myFunction2() {

  let x = "Hello";

  //console.log(y);

  function anotherFunction() {

  console.log(x);

    // const y = 10;

  }

  anotherFunction();

}
```

Importance of Scope in js

The main reasons for having different scopes in programming languages are:

- To allow programming in large teams where different people work on different parts of the code. By hiding some parts of the code (encapsulation), team members working on other parts of the code are less likely to accidentally break each other's code.

- For performance reasons. It takes time for a computer to look up a variable in an outer scope - this is called variable lookup overhead. By limiting the amount of code in the same scope, you can reduce that overhead.

- For security reasons. By making some parts of your code private, you can stop people from accidentally or maliciously breaking your code

High Order function

High order function vs function as first class function in javascript

A function is a first-class function if it can be used like any other value in the language. This means that the function can be:

A function is a higher order function if it takes one or more functions as arguments or returns a function.

1. They are objects that can be passed as arguments to other functions.

2. They can be returned by other functions.

3. They can be assigned to variables.

4. They can be stored in arrays.

Functions returning functions

create a function that returns a new function

```
function addTwoNumbers(a) {

  return function (b) {
    return a + b;
  };
}
//call the function
const myAnswer = addTwoNumbers(2);
console.log(myAnswer(5));
//or
const myAnswer2 = addTwoNumbers(2, 5)(2, 5);
```

Assigning a function to a variable

```
function sayHello() {
  console.log("Hello");
}
const myFn = sayHello;

//assign the function to a variable
const sayHello2 = function () {
  console.log("Hello");
};
```

stored funtions in an arrays

```
function sayHello() {
  console.log("Hello");
}
```

```
const sayHello2 = function () {
  console.log("Hello");
};


const array = [sayHello, sayHello2];
//call the function in the array
array[0]();
array[1]();
```

High Order Function (HOF)

Functions accepting functions as arguments(Higher Order Functions or Callback Functions)

```
calculate the billconst calBill = function (quantity, price) {
  return quantity * price;
};

const displayBill = function (calBillFn) {
  console.log(`Your bill is ${calBillFn(2, 5)}`);
};

/another of calling the function
const displayBill2 = function (calBillFn) {
  return calBillFn;
```

};

/all the function

const ans = displayBill(calBill);

Functions returning functions

Create a function that returns a new function

```
function addTwoNumbers(a) {
  return function (b) {
    return a + b;
  };
}
```

call the function

```
const myAnswer = addTwoNumbers(2);
console.log(myAnswer(5));
```

//or

```
const myAnswer2 = addTwoNumbers(2, 5)(2, 5);
```

Default paramenters

ES5

```
function addComment(name, comment) {
  name = name || "Anonymous";
  comment = comment || "No comment";
  return `${name} says: ${comment}`;
}
```

ES6
```
function addComment(name = "Anonymous", comment = "No comment") {

  return `${name} says: ${comment}`;

}
```

Default parameters with expressions

ES6
```
function totalBill(base, tax = base * 0.13) {

  return base + tax;

}
```

//es5
```
function totalBill(base, tax) {

  if (tax === undefined) {

    tax = base * 0.13;

  }

  return base + tax;

}
```

Skip the default parameter when calling the function

ES6
```
function makePost(user = "Anonymous", title, body) {

  return {

    user,

    title,
    body,
  };
}
makePost(undefined, "Post title", "Post body");
```

A high-level overview of JS Part 1

History of javascript?

JavaScript was created in 1995 by Brendan Eich while working for Netscape Communications. It was originally named Mocha, but was later renamed to LiveScript, and then finally to JavaScript.

LiveScript was renamed JavaScript in the same year, and the language has been known by that name ever since. In 1996, Microsoft released Internet Explorer 3.0 with support for JavaScript.

In the years that followed, numerous other browsers were released with built-in JavaScript support, and the language became increasingly popular as a way to add dynamic content to websites. In 1997, JavaScript was standardized by ECMA International in the ECMAScript language specification.

In the early 2000s, Ajax and other web technologies were developed that use JavaScript to create dynamic, responsive web applications. Today, JavaScript is one of the most popular programming languages on the web and is used for a wide variety of tasks including front-end web development, server-side scripting, and game development.

Today, JavaScript is one of the most popular programming languages in the world and is used for a wide variety of tasks including web development, application development, and game development.

JavaScript's syntax takes after C++ and Java quite a bit. If you have experience with either of those languages, JavaScript's syntax will look familiar. That being said, JavaScript functions more like a dynamically-typed, interpreted language like Python or Ruby.

JavaScript is an interpreted language, not a compiled language. A program such as C++ or Java needs to be compiled before it is run.

The source code is passed through a program called a compiler, which translates it into bytecode that the machine understands and can execute. In contrast, JavaScript has no compilation step. Instead, an interpreter in the browser reads over the JavaScript code, interprets each line, and runs it. More modern browsers use a technology known as Just-In-Time (JIT) compilation, which compiles JavaScript to executable bytecode just as it is about to run.

JavaScript is named after Java, and many of its concepts are borrowed from the Java language. However, Java and JavaScript are two entirely distinct languages. The most significant difference between them is that Java is a compiled language, and JavaScript is an interpreted language. JavaScript runs on many browsers out-of-the-box, whereas Java applets require an additional plug-in. Both languages have different runtime environments, different governing bodies, and different libraries.

While JavaScript has its flaws, it is still a very useful language. It runs in every web browser and can be used to write cross-platform applications. Additionally, platforms like Node.js allow developers to run JavaScript on the server side. This means that it is possible to create entire web applications using JavaScript.

JavaScript can be a complex language, and most teams only use a fraction of JavaScript's capabilities. If you consult a style guide, it will recommend specific JavaScript techniques, constructs, and libraries.

javascript Console.log()

JavaScript's console.log() function is used to print text to the console. It can be used to print simple text, variables, objects, and arrays. The console.log() function is typically used for debugging purposes.

JavaScript is most frequently executed on webpages inside the browser, but it can also be run server-side. For now, we will run JavaScript in the console, which will enable us to see the results of our code more rapidly.

The beauty of interpreted languages is that they are designed to be executed in a single pass through the source code, running each instruction step-by-step. That means that we can provide the interpreter with a single step and request that it runs it.

JavaScript consoles were built inside browsers. It is like a command-line interface that runs JavaScript on your JavaScript engine.

Behind the console is the read-eval-print loop (REPL). This refers to the loop that the console runs: it first reads your input, then it evaluates it as JavaScript code, then it prints the results. You will sometimes hear the term REPL being used to refer to any sort of programming shell that allows you to enter code and see results immediately. For instance, Nodejs, Python and Ruby also provide similar REPL shells.

Try running some simple math expressions within the console, such as 1 + 2 or 3 * 4. JavaScript prints the correct answers back at you. Congratulations, you can now use your computer as a very expensive calculator.

When executing a function in the console ex using the console.log(). It produces two lines of output. The first line is the results that we expect. The second line reads undefined. This is because the first line is the output that we instructed JavaScript to print, and the second line is the result of evaluating our program. Every JavaScript expression has a result, but some expressions, such as the console.log function, return an empty result called undefined.

Features of javascript?

- It's a garbage collector
- It's a non-blocking and event loop language
- It's a object-oriented language using prototypes
- It's a functional language

- It's a high-level language (resources to manage applications)
- It's a dynamically typed language
- It's a single threaded language
- It's an interpreted language (not compiled)
- It's a client-side language
- It's a server-side language (Node.js runtime)
- It uses first-class functions
- It's a single threaded language

Single threaded language

- This means that js executes the code line by line.
- This means that the code is executed one by one.
- This means that the code is executed sequentially.
- This means that the code is executed in the same order.

Example

```
let myName = "John";

let age = 30;

const displayName = () => {

  console.log(myName);

};

displayName();

const sayHello = () => {

  console.log("Hello");

};
```

```
sayHello();
```

Javascript is a dynamically typed language

This means that the type of a variable can change during the program execution. For example, if we assign a string to a variable, the type of the variable will be string.

With this, we can use the same variable to store different types of data.

There are no restrictions on the type of data that can be stored in a variable. We can store any type of data in a variable, unlike in C++.

If you need a strict type checking, you can use typeof operator or you can use typescript.

```
Example

let a = "string";

let b = 10;

b = true;
```

Object Oriented Programming

It's a programming paradigm that uses objects as the primary way of representing data in a program which uses Prototypal Inheritance

Prototypes are used to create new objects that inherit properties and methods from another object.

Example using arrays methods

Array prototype

```
//----
Array.prototype.push("element");

let arr = [1, 2, 3, 4, 5];

arr.push(6);
arr.pop();

arr.shift();
```

Example using objects methods

Object prototype

1. Object.keys(obj)

2. Object.values(obj)

object hasOwnProperty(property)

It's a method that checks if an object has a property or not.

```
Object.prototype.hasOwnProperty("property");

let obj = {

 name: "John",

 age: 30,

 isMarried: false,

};

obj.hasOwnProperty("name");

const ans = obj.hasOwnProperty("isMarried");
```

It's a process that frees up memory by removing objects that are no longer used in the program and it's called garbage collection.

It's a code that runs concurrently with the code that is running in the background. This means that the code that is running in the background is not blocked by the code that is running in the foreground.

But js is a single threaded language. This is possible because of the nature of the event loop.

Example of non-blocking using setTimeout

```
const sayHello2 = () => {

  console.log("Hello");

};

setTimeout(sayHello, 2000);

console.log("Hi"); //This code will be executed before the code
in the setTimeout function.
```

High-level language

Any applications uses resources like memory, CPU, network, etc. to be run on a computer.

For low level programming languages like C, C++, Java, etc. developers need manage these resources manually.

For high level programming languages like JavaScript, Python, Ruby, etc. resources management are done automatically.

JS is a multi paradigm language

This a coding approach to structure code

//Example

1. Functional programming

2. Object oriented programming

3. Procedural programming

4. Declarative programming

5. Imperative programming

6. Hybrid programming

Javascript Engine

Javascript engine is a computer program that runs javascript code.

Computers only understand machine code.

Chrome: uses V8 engine

Firefox: uses SpiderMonkey engine

Google Chrome - V8

Mozilla Firefox - SpiderMonkey

Internet Explorer/Edge - Chakra Opera - Presto (discontinued)

Code execution in js

source code -> parser -> AST(abstract syntax tree) -> Compilation(convert AST code) -> machine code -> interpreter -> browser

sourc code: It's the code that is written by the user.

parser: It's a program that converts the source code to an abstract syntax tree.

AST: It's a tree that represents the code.

Compilation: It's a program that converts the AST code to machine code.

machine code: It's the code that is executed by the computer.

interpreter: It's a program that executes the machine code by reading it line by line.

browser: It's a program that executes the interpreter(display the result to user).

//Since js is not a compiled language, it doesn't have a compiler but instead modern browsers uses what is called Just-In-Time (JIT) compilation, which compiles JavaScript to executable bytecode just as it is about to run.

Is it possible, to view the final machine code?

Your script doesn't transform to machine code directly. JavaScript runs on virtual machine V8 (it's true for chrome and classic nodejs) and you can get VM byte code using:

ANSWER

Your script doesn't convert to machine code directly. JavaScript operates on virtual machine V8 (this is true for chrome and nodejs) and you can obtain VM byte code by using: node --print-bytecode script.js

JS Runtime (Browser)

JS runtime, in a laymans point of view, it's a box that contains all the things we need to run our code.

EXAMPLE

1. The browser

2. The console

3. The engine (V8) (callstack, heap)

4. Web APIs (DOM, Storage, Network,setTimeout, setInterval,)

5 **Callback Queue**(contains all the callbacks that are waiting to be executed eg: setTimeout, setInterval, button click, when the button is clicked, the callback is added to the callback queue, when the stack is empty, the callback is executed and this uses a technique called the event loop.

The event loop takes callbacks from the callback queue and puts them in the stack.)

6. **Event Loop**: This helps us to implement non-blocking code and concurrent code.

7 . **WEB APIs** are the APIs that are provided by the browser, it's not part of the JavaScript language but it adds more functionality to the JavaScript engine.

Call Stack in js

A call stack is a data structure that is used to track the execution of function calls. It is used to keep track of the order in which function calls are made.

When a function is called, it is added to the top of the call stack. The function then gets executed and, when it is finished, it is removed from the top of the stack. This process continues until the call stack is empty and all of the functions have been executed.

Asynchronous and Synchronous programming in javascript

There are two types of code - synchronous and asynchronous. Synchronous code is executed in a linear fashion, one line at a time. Asynchronous code, on the other hand, can be executed out of order or in parallel.

Both synchronous and asynchronous code have their benefits and drawbacks. Synchronous code is typically simpler to write and understand, but it can be slower to execute. Asynchronous code can be more complex, but it can also be faster and more scalable.

The key distinctions between asynchronous and synchronous processes are as follows: Asynchronous processes are multi-threaded, meaning that multiple operations or programs can execute concurrently. Conversely, synchronous processes are single-threaded, so only one operation or program can run at a time. Additionally, asynchronous processes are non-blocking, meaning that they can send multiple requests to a server without waiting for a response.

setTimeout function in javascript

The setTimeout() function is used to set a delay for a certain action. It can be used to delay an action by a set amount of time or to delay an action until a certain point in time. The setTimeout() function is part of the window object and can be used with any other object.

The setTimeout() function takes two arguments. The first argument is the code to be executed. The second argument is the delay in milliseconds. The code to be executed can be a function or a string. If the code is a function, it will be executed with the same this object as the setTimeout() function. If the code is a string, it will be executed as if it were a part of the setTimeout() function.

The setTimeout() function will return a numeric id that can be used to cancel the delay.