**What is Java Script?**

JavaScript is a lightweight programming language

commonly used by web developers to add dynamic interactions to web pages, applications, servers, and even games.

It works seamlessly alongside HTML and CSS, complementing CSS in formatting HTML elements while providing user interaction, a capability that CSS alone lacks.

Ecma International (formally European Computer Manufacturers Association)

.

**Ecma International (formally European Computer Manufacturers Association)**

After Netscape submitted it to **ECMA International** as a standard specification for web browsers, JavaScript pioneered the release of ECMAScript.it was published in 2015, and is also known as ECMAScript 2015.

**What Is JavaScript Used For?**

### Web and Mobile Apps

### Building Web Servers and Server Applications

### Interactive Behaviour on Websites

### Game Development

**What Makes JavaScript Great?**

JavaScript has several advantages that make it a better choice than its competitors.

The following are several benefits of using JavaScript:

* **Simplicity** ‒ having a simple structure makes JavaScript easier to learn and implement, and it also runs faster than some other languages. Errors are also easy to spot and correct.
* **Speed** ‒ JavaScript executes scripts directly within the web browser without connecting to a server first or needing a compiler. Additionally,

most major browsers allow JavaScript to compile code during program execution.

* **Versatility** ‒ JavaScript is compatible with other languages like PHP, Perl, and Java. It also makes data science and machine learning accessible to developers.
* **Popularity** ‒ plenty of resources and forums are available to help beginners with limited technical skills and knowledge of JavaScript.
* **Server load** ‒ another perk of operating on the client-side is that JavaScript reduces the requests sent to the server. Data validation can be done via the web browser, and updates only apply to certain web page sections.
* **Updates** ‒ JavaScript development team and ECMA International continuously update and create new frameworks and libraries, ensuring its relevance within the industry.

**Promise**: an object that may or may not contain some value at some point in the future.

A promise is just some set of code that says you are waiting for some action to be taken and then once that action is complete you will get some result. Sometimes, though, a promise will be unfulfilled and you will not get the result you expect and instead will receive a failure/error.

**Parameters**

* The promise constructor takes only one argument which is a callback function
* The callback function takes two arguments, *resolve* and *reject*
  + Perform operations inside the callback function and if everything went well then call resolve.
  + If desired operations do not go well then call reject.

**A Promise has four states:**

1. **fulfilled**: Action related to the promise succeeded
2. **rejected**: Action related to the promise failed
3. **pending**: Promise is still pending i.e. not fulfilled or rejected yet
4. **settled**: Promise has been fulfilled or rejected

Example

letpromise = newPromise(function (resolve, reject) {

    constx = "Ps-Softech";

    consty = "Ps-Softech"

    if (x === y) {

        resolve();

    } else {

        reject();

    }

});

promise.then(function () {

        console.log('Success, You are a GEEK');

    }).catch(function () {

        console.log('Some error has occurred');

    });

}

Example 2

let promise = new Promise(function (resolve, reject) {

resolve('Ps-Softech');

})

promise.then(function (successMessage) {

//success handler function is invoked

console.log(successMessage);

}, function (errorMessage) {

console.log(errorMessage);

});

Example 3

let promise = new Promise(function (resolve, reject) {

reject('Promise Rejected')

})

promise.then(function (successMessage) {

console.log(successMessage);

}, function (errorMessage) {

//error handler function is invoked

console.log(errorMessage);

});

Example 4

let promise = new Promise(function (resolve, reject) {

reject('Promise Rejected')

})

promise.then(function (successMessage) {

console.log(successMessage);

})

.catch(function (errorMessage) {

//error handler function is invoked

console.log(errorMessage);

});

**Introduction to JavaScript closures**

In JavaScript, a closure is a function that references variables in the outer scope from its inner scope. The closure preserves the outer scope inside its inner scope.

To understand the closures, you need to know how the lexical scoping works first.

Lexical scoping

Lexical scoping defines the scope of a variable by the position of that variable declared in the source code. For example:

Example

let name = 'John';

function greeting() {

let message = 'Hi';

console.log(message + ' '+ name);

}

In this example:

* The variable name is a global variable. It is accessible from anywhere including within the greeting() function.
* The variable message is a local variable that is accessible only within the greeting() function.

If you try to access the message variable outside the greeting() function, you will get an error.

So the JavaScript engine uses the scope to manage the variable accessibility.

According to lexical scoping, the scopes can be nested and the inner function can access the variables declared in its outer scope. For example:

Example

function greeting() {

let message = 'Hi';

function sayHi() {

console.log(message);

}

sayHi();

}

greeting();

The greeting() function creates a local variable named message and a function named sayHi().

The sayHi() is the inner function that is available only within the body of the greeting() function.

The sayHi() function can access the variables of the outer function such as the message variable of the greeting() function.

Inside the greeting() function, we call the sayHi() function to display the message Hi

Example

function greeting() {

let message = 'Hi';

function sayHi() {

console.log(message);

}

return sayHi;

}

let hi = greeting();

hi(); // still can access the message variable

Now, instead of executing the sayHi() function inside the greeting() function, the greeting() function returns the sayHi() function object.

Note that functions are the first-class citizens in JavaScript, therefore, you can return a function from another function.

Outside of the greeting() function, we assigned the hi variable the value returned by the greeting() function, which is a reference of the sayHi() function.

Then we executed the sayHi() function using the reference of that function: hi(). If you run the code, you will get the same effect as the one above.

However, the interesting point here is that, normally, a local variable only exists during the execution of the function.

It means that when the greeting() function has completed executing, the message variable is no longer accessible.

In this case, we execute the hi() function that references the sayHi() function, the message variable still exists.

The magic of this is closure. In other words, the sayHi() function is a closure.

A closure is a function that preserves the outer scope in its inner scope.

Example

functiongrandParent() {

  varhouse = "GreenHouse";

  functionparent() {

    varcar = "Tesla";

    house = "YellowHouse";

    functionchild() {

      varscooter = "Vespa";

      console.log("I have:", house, car, scooter);

    }

    returnchild;

  }

  returnparent;

}

varP= grandParent();

//varH=P()

//H()

# **What is function hoisting in JavaScript?**

JavaScript has a feature that allows function and variables to be hoisted – this is useful for many developers. **Hoisting** is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution.

## Variable Hoisting

Variables are not hoisted as they cannot be used before their declaration. As shown by the code below, the variable x was used before it was declared. At this point, the variable x does not exist. Therefore, an error is thrown when you try to access the value of a variable that doesn’t exist. The value returned by x is undefined, as illustrated by the output:

Example

console.log(x);

var x = "Educative is amazing!";

console.log(x);

## Function hoisting

Hoisted functions can be used before their declaration. As you can see, the JavaScript interpreter allows you to use the function before the point at which it was declared in the source code. This is extremely useful as the function can be called before defining it. One can then define the function anywhere in the program code. As can be seen from the code below, the Func\_Hoisted() function is called before it is declared. Function declarations can be hoisted.

Example

Func\_Hoisted();

function Func\_Hoisted() {

console.log("This function is hoisted!");

}

## Function expressions vs. function declarations

Function definition hoisting only occurs for function declarations, not function expressions. Function expressions are defined with a function as a variable. The definitions look like var b = function(). This kind of function declarations through expressions makes the function NOT Hoisted. As can be seen in the example below, function a is defined as a declaration, but function b is defined as an expression. So, when we call function b, it gives an error claiming that function b is not defined.

Example

// This will work

a();

// This will throw an error

b();

// function declaration

function a() {

console.log("This function is hoisted!");

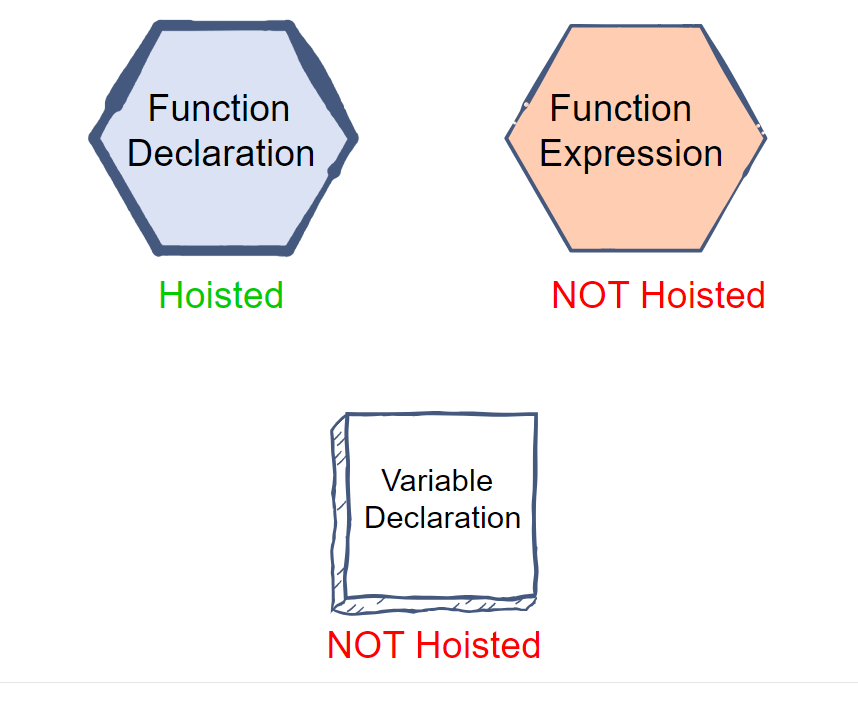
}

// function expresssion

var b = function () {

console.log("This declaration makes function not hoisted!");

}



**let & var**

The var variables belong to the global scope when you define them outside a function. For example:

var counter;

In this example, the counter is a global variable. It means that the counter variable is accessible by any functions.

When you declare a variable inside a function using the var keyword, the scope of the variable is local. For example:

functionincrease() {

var counter = 10;

}

*// cannot access the counter variable*

In this example, the counter variable is local to the increase() function. It cannot be accessible outside of the function.

The following example displays four numbers from 0 to 4 inside the loop and the number 5 outside the loop.

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

console.log("Inside the loop:", i);

}

console.log("Outside the loop:", i);

Output:

Inside the loop: 0

Inside the loop: 1

Inside the loop: 2

Inside the loop: 3

Inside the loop: 4

Outside the loop: 5

Code language:Shell Session(shell)

In this example, the i variable is a global variable. Therefore, it can be accessed from both inside and after the for loop.

The following example uses the let keyword instead of the var keyword:

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

console.log("Inside the loop:", i);

}

console.log("Outside the loop:", i);

In this case, the code shows four numbers from 0 to 4 inside a loop and a reference error:

Inside the loop: 0

Inside the loop: 1

Inside the loop: 2

Inside the loop: 3

Inside the loop: 4

The error:

Uncaught ReferenceError: i is not

Since this example uses the let keyword, the variable i is blocked scope. It means that the variable i only exists and can be accessible inside the for loop block.

In JavaScript, a block is delimited by a pair of curly braces {} like in the if...else and for statements:

if(condition) {

*// inside a block*

}

for(...) {

*// inside a block*

}

## #2: Creating global properties

The global var variables are added to the global object as properties. The global object is window on the web browser and global on Node.js:

var counter = 0;

console.log(window.counter); *//*

However, the let variables are not added to the global object:

let counter = 0;

console.log(window.counter);

## #3: Redeclaration

The var keyword allows you to redeclare a variable without any issue:

var counter = 10;

var counter;

console.log(counter); *//* )

However, if you redeclare a variable with the let keyword, you will get an error:

let counter = 10;

let counter; *//*

## #4: The Temporal dead zone

The let variables have temporal dead zones while the var variables don’t. To understand the temporal dead zone, let’s examine the life cycles of both var and let variables, which have two steps: creation and execution.

### The var variables

* ***In the creation phase, the JavaScript engine assigns storage spaces to var variables and immediately initializes them to undefined.***
* In the execution phase, the JavaScript engine assigns the var variables the values specified by the assignments if there are ones. Otherwise, the var variables remain undefined.

See the execution context for more information.

### The let variables

* ***In the creation phase, the JavaScript engine assigns storage spaces to the let variables but does not initialize the variables. Referencing uninitialized variables will cause a ReferenceError***.
* The let variables have the same execution phase as the var variables.

The temporal dead zone starts from the block until the let variable declaration is processed. In other words, it is the location where you cannot access the let variables before they are defined.

**Difference Between Function & Arrow Function**

**What is a Function?**

Functions are one of the building blocks of JavaScript. A function in JavaScript is just like a procedure—a set of statements that perform a task or calculate a value. Still, to qualify as a function, a procedure should take certain inputs and return some output where there is some apparent relationship between the input and output. If you want to use a function, you have to define it somewhere in the scope from which you want to call it.

**Normal Function**

To create a normal function, we use the "function" keyword followed by an assigned name. Then we create an area where user input will determine outcomes. And the output will be based on sets of instructions provided within brackets.

As we can see, this is an example of normal function. And this function accepts two values and prints their sum:

//an addition function written in javascript

function addition(x, y) {

 return x + y;

}

We can call this function by its name:

let answer = addition(9, 18);

console.log(answer); // Output: 27

**Arrow Function**

Arrow functions use the "fat arrow" syntax (=>). Arrow functions offer a more concise way of constructing functions. Arrow functions were introduced with ES6, and they provide a simpler way of writing functions.

The arrow function expression is an alternative to the normal function. With some differences and limitations in usage:

* Arrow functions don't have their bindings to arguments, this, or super, and we cannot use them as methods(member).
* Arrow functions cannot be used as constructors. Calling them with new throws a TypeError. They also don't have access to the new.target keyword.
* Arrow functions cannot use yield within their body and cannot be created as generator functions.

The below function is similar to the previous function, written as an arrow function:

const addition = (x, y) => x + y; //written in javascript.

A key aspect of arrow functions is their concise and streamlined syntax. With parentheses containing the arguments, a fat arrow (=>), then a function body, it's easy to see how simple they can be. For this example, the function body contains just one expression returning a sum of two parameters. As a result, both curly braces and the return keyword will not be needed.

To call this function, you would use the same syntax as before:

let answer = addition(9, 18);

console.log(answer); // Output: 27

Having examined the syntax of both function types, We will now dive into the divergences between them.

**Differences Between Normal Function and Arrow Function**

| **Aspect** | **Arrow functions** | **Normal functions** |
| --- | --- | --- |
| **Anonymous** | Yes | No |
| **Syntax** | Shorter | Longer |
| **Handling multiple expressions** | Difficult | Better |
| **"this" keyword behavior** | Inherits from a broader scope. | Refers to the object it belongs to ("this"). |
| **Constructor functionality** | Not suitable for use as a constructor. | Suitable for use as a constructor. |

**To learn about these differences in detail, read the following.**

* **Arrow functions are anonymous:**

Arrow functions are unnamed. It means that they do not have a name and cannot have designated identifiers. Instead, they can be linked with variables or transferred as an argument for another function.

The arrow function can also be assigned to a variable via the code given below:

const addition = (x, y) => x + y; //written in javascript

In the above example, the variable name (addition) is used to refer to the function.

* **Arrow functions have a shorter syntax:**

Arrow functions have a shorter syntax. Making them useful in writing one-line blocks of code. Normal functions are better at handling logic that requires multiple expressions.

E.g., of the normal function that calculates the factorial of a number:

//written in javascript

function factorial(n) {

 if (n === 0) {

   return 1;

 } else {

   return n \* factorial(n - 1);

 }

}

A function that uses recursion for calculating the factorial of a number is what we need. The function should have curly braces and the return keyword, as it contains multiple statements.

Here is the same function written as an arrow function:

//written in javascript

const factorial = (n) => {

 if (n === 0) {

   return 1;

 } else {

   return n \* factorial(n - 1);

 }

}

As this function contains various statements. And it cannot be represented as a single expression. The arrow function syntax seems difficult and lengthy.

* **Absence of "this" value in the Arrow function:**

In the case of the arrow function, they do not have their own "this" value. Traditional methods refer to objects they belong to as "this," while arrow methods inherit their context from a broader scope. This can cause issues when using arrow functions with objects.

For example,

//written in javascript

const human = {

 name: 'Sourabh',

 age: 21,

 sayName: function() {

   console.log(this. name);

 }

};

With our current example, there exists an item referred to as "human".And this has properties such as a given name and age. Alongside these, a function titled "sayName" exists for this individual. It shall be produced on the console whenever it calls its name using the "this" method.  
  
Below is the sayName method written with an arrow function:

//written in javascript

const human = {

 name: 'Sourabh',

 age: 21,

 sayName: () => {

   console.log(this. name);

 }

};

If we call the sayName function at present, then it would show "undefined" as a result on the console. This occurs because the "this" keyword doesn't indicate the person's object. Rather it points to the global object process (which is none other than a window object. Particularly in web browsers).

* **Arrow functions cannot be used as constructors:**

Not being able to work as the constructor is the main difference between the arrow and regular functions. As we know, Constructors create new object instances uniquely.

For example:

//written in javascript

function Human(name, age) {

 this.name = name;

 this.age = age;

}

constsourabh = new Person('Sourabh', 21);

console.log(sourabh.name); // Output: Sourabh

In this example, a  functional object known as a Person is used to assign two distinct parameters. And these will be used in the formation of an item. The name and age specifications are set for the new creation by the Person function used. A fresh article named sourabh is produced by using a similar format that is used for generating a new instance of an item. Finally, his name will be displayed on the console.

Below we have written the Person function as an arrow function:

//written in javascript

const Person = (name, age) => {

 this.name = name;

 this.age = age;

}

const john = new Person('Sourabh', 21); // Error!

When we try to create a new object from the Person class, it will result in failure since arrow functions are not suitable for use as constructors. This limitation is responsible for the error message.