

**Function declaration vs Expressions**:

Consider this piece of code,

// Function declaration

function walk(){

  console.log("walk")

}

This is an example of a ***function declaration***.

But in JavaScript we have another way to define a function,

let run = function(){};

This is what we call a ***function expression***. ***“****Just like we can set a variable to a number or a string, similarly we can set it to a function as well****”***.

Since in JavaScript functions are objects, therefore setting the *run* variable to a function is similar to setting it to an object.

Note: By convention we put semicolon ; at the end of function expression not at end of function declaration. This is also one of the difference between function declaration and expression.

let run = function () { //***run*** variable acts as a reference

  console.log("run"); // to this anonymous function

};

Here basically we have defined a function and this function does not have a name. Its syntax is very much similar to what we have in function declaration and we call it an ***anonymous function expression***.

Now we can call this anonymous function using its reference which is *run* variable.

let run = function () {

  console.log("run");

};

run(); //run

same way like we call a function in JavaScript.

We can declare a new variable called *move* and set it to *run*.

let run = function () {

  console.log("run");

};

let move = run;

run(); //run

move(); //run

Now both *move* and *run* acts as a reference to this anonymous function which is one object in memory and can be called using its references.

***Recap***:

In JavaScript we have to ways of defining a function we can either use function declaration syntax or function expression which basically involves declaring a variable or a constant and setting it to a function.

**Hoisting**:

There is a major key difference between function declaration an function expression

We have a *walk*() function which is a function declaration, so if I call it before it is defined, what will happen?

walk(); //walk

function walk() {

  console.log("walk");

}

But if I try to do the same with my *run* function expression,

run();

let run = function () {

  console.log("run");

};

I get ReferenceError: Cannot access 'run' before initialization

*Why is that*?

It is because **“***when our* *JavaScript engine executes our code, it moves all the* ***function declarations*** *to the top****”***.

function walk() {

  console.log("walk");

}

//This is what it looks at runtime.

walk();

This is called *hoisting*.

***“****It is the process of moving function declarations to the top of the file which is automatically done by JavaScript engine that is executing this code****”***.

**Arguments**:

As we learned earlier that JavaScript is a dynamic language, so we can declare a variable, set it to a number then change its type and set it to a string.

let x = 1;    //number

x = "a";      //string

We have the same concept in the arguments of a function, Consider this example of *sum* function.

function sum(a, b) {

  return a + b;

}

console.log(sum(1, 2)); //3

If instead of passing two arguments we pass only a=1,

console.log(sum(1)); //NaN

We get NaN which is short for not a number. This happened because a = 1 and b = undefined in this case, therefore

*1+ undefined* = NaN.

Similarly we can pass multiple arguments instead of 2, which is perfectly fine JavaScript expression,

console.log(sum(1, 2, 3, 4, 5)); //3, first two arguments used.

*What if we want flexibility of passing as many arguments as want and get their sum***?**

Every function in JavaScript has a special object called *arguments*.

function sum(a, b) {

  console.log(arguments);//[Arguments] { '0': 1, '1': 2, '2': 3, '3': 4, '4': 5 }

  return a + b;

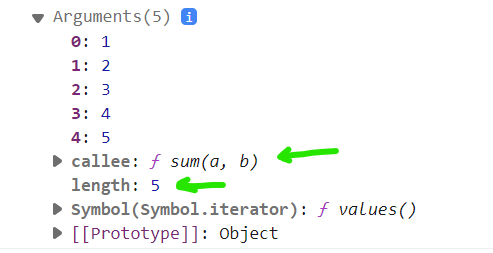
}

console.log(sum(1, 2, 3, 4, 5)); //3

Here we see this arguments object,

[Arguments] { '0': 1, '1': 2, '2': 3, '3': 4, '4': 5 }

The keys ‘0’, ‘1’, ‘2’ are the indexes of the arguments that we passed to this function.



This object has *length* property that returns number of arguments that are passed. The other property is *callee* which returns currently executing function.

If you want to have a function with varying number of parameters or to work with all the arguments that are passed to this function we use this ***arguments*** object.



Notice that this *arguments* object has an iterator, so we can easily iterate over it using a for – of loop,

function sum() { //Now, we do not need parameters here

  let total = 0;

  for (let value of arguments) {

    total += value;

  }

  return total;

}

console.log(sum(1, 2, 3, 4, 5)); //15

**The Rest Operator**:

In modern JavaScript if you want to have function with a varying number of parameters you can use the ***rest operator***.

function sum(...args) {}

we use *…args* syntax as a parameter of this function.

Note: Do not confuse ***rest*** with *spread* operator because spread operator looks exactly the same. *We use* ***spread*** *operator with arrays to spread its individual elements.*

But when we use this **…** with parameters of the function, it becomes a *rest* operator.

function sum(...args) {

  console.log(args); // [ 1, 2, 3, 4, 5, 10 ]

}

console.log(sum(1, 2, 3, 4, 5, 10));

args is an array of 6 elements.

Now we can easily use reduce method here to get sum,

function sum(...args) {

  return args.reduce((acc, value) => acc + value);

}

console.log(sum(1, 2, 3, 4, 5, 10)); //25

This is more elegant solution for this problem and is also flexible in case we need some more calculations in this method,

For example:

Let us add a discount parameter in our *sum* function to apply on the total sum of item prices in a shopping cart.

function sum(discount, ...prices) {

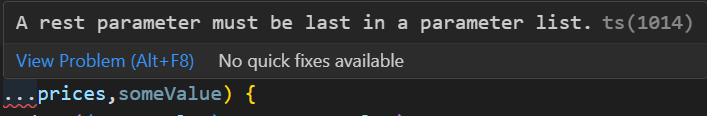
  const total = prices.reduce((acc, value) => acc + value);

  return total \* (1 - discount);

}

console.log(sum(0.1, 20, 30)); //45

Note: If we add one more parameter after rest parameter, we will get a Syntax error.



So rest parameters must always be the last parameter to be declared in a function definition.

That is why it is called Rest parameter, because all the other parameters are before and *rest of the parameters are at last*.

**Default Parameters**:

There are times when we want to supply default values to the parameters of a function.

Take this interest calculating function as example,

function interest(principal, rate, years) {

  return ((principal \* rate) / 100) \* years;

}

console.log(interest(10000, 3.5, 5)); //1750

We can set some default values for rate and years in our function, if user does not provide them using || operator.

function interest(principal, rate, years) {

  rate = rate || 3.5;

  years = years || 5;

  return ((principal \* rate) / 100) \* years;

}

console.log(interest(10000)); //1750

Starting from ES6 we have a cleaner way to achieve the same thing by *setting the default values right where we declare parameters*.

function interest(principal, rate = 3.5, years = 5) {

  return ((principal \* rate) / 100) \* years;

}

console.log(interest(10000)); //1750

There is just one small caveat here, after we give a parameter a default value, we should give all the parameters afterward also a default value or otherwise we will get NaN.

Here we removed default value for years,

function interest(principal, rate = 3.5, years) {

  return ((principal \* rate) / 100) \* years;

}

console.log(interest(10000)); //NaN

In this case years is undefined.

*Rule of thumb*,

Whenever you want to give a function parameter a default value make sure that parameter is the last parameter in the list or all the parameters after that have a default value.

**Getters and Setters**:

We have some special kind of methods in an object called getters and setters.

Let us say we have a *person* object with these two properties,

const person = {

  firstName: "Himanshu",

  lastName: "Pandey",

};

Now somewhere in our application, we want to display that person’s full name which we can do by using template literal,

console.log(`${person.firstName} ${person.lastName}`);

But there can be a limitation in this approach like this same information we might have to display at multiple locations in our app.

We do not have to repeat this same expression in multiple places.

*Better approach*:

Define a method in this object, call it full name and move our expression inside it and whenever we want to display a person’s full name, call that function.

const person = {

  firstName: "Himanshu",

  lastName: "Pandey",

  fullName() {

    console.log(`${person.firstName} ${person.lastName}`);

  },

};

person.fullName(); //Himanshu Pandey

However there are couple of problems with this approach,

First one is that person.fullName is read only. In other words we cannot set a person’s full name from outside.

person.fullName = "John Smith" //not possible

It would be nice if we could do that.

Second, we should be able to call this like a property not a function,

person.fullName; //not person.fullName();

***“****We use* ***getters*** *for accessing properties in an object and* ***setters*** *in order to change or mutate them****”*.**

To convert our function into a getter, we will add ***get*** keyword before function name.

const person = {

  firstName: "Himanshu",

  lastName: "Pandey",

  get fullName() {

    console.log(`${person.firstName} ${person.lastName}`);

  },

};

person.fullName; //Himanshu Pandey

Now we are calling *fullName* like a property not as a function.

Now if we want to set firstName and lastName from outside, we need to create a *setter* function similar syntax just using ***set*** keyword.

const person = {

  firstName: "Himanshu",

  lastName: "Pandey",

  get fullName() {

    console.log(`${person.firstName} ${person.lastName}`);

  },

  set fullName(value) { //value we get from user(required parameter)

    let parts = value.split(" "); //splitting john “ ” smith

    this.firstName = parts[0]; //assigning 0th index value to first Name

    this.lastName = parts[1]; //assigning 1st index value to last Name

  },

};

person.fullName = "John Smith";

person.fullName; //John Smith

If you log the *person* object on console,



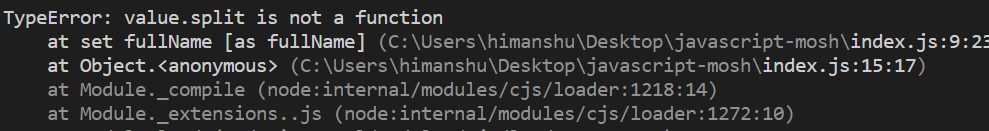
**Try and Catch**:

In last example we assumed that the *value* that we received in *fullName* was a valid string.

What would happen if we pass a Boolean here,

person.fullName = true;

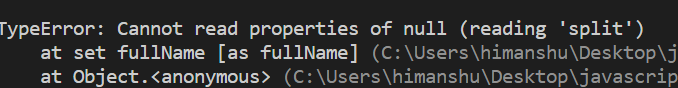
we get TypeError:



because *split* is a method that belongs to strings.

Similarly if we pass null or undefined here, we get error,

person.fullName = null;



So we need to introduce some error handling in our code, typically in situations like this we should *do error handling at the beginning of a function or a method*. This is called ***defensive programming***.

const person = {

  firstName: "Himanshu",

  lastName: "Pandey",

  set fullName(value) {

    if (typeof value !== "string") return; //if type of value not string, return

    let parts = value.split(" ");

    this.firstName = parts[0];

    this.lastName = parts[1];

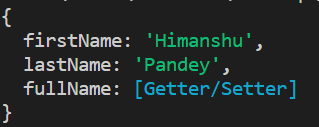
  },

};

person.fullName = null;

console.log(person);

We did not get any error now and default values are obtained.



But sometimes we need to report an error in our application, that is where we throw an ***exception***. So instead of returning in this statement,

if (typeof value !== "string") return;

we replace return with *throw* and create a *new Error* object and pass an error message as its argument.

  set fullName(value) {

    if (typeof value !== "string") throw new Error("value is not a string");

As we know, *Error() is a constructor function and we are calling this function using new operator to create a new Error object*.

Note: There is a slight difference in error and exception in JavaScript. We can define an error object using Error constructor function but the moment we *throw* it, it becomes an exception.

const e = new Error(); //an error object

throw e; //it becomes an exception now.

Now we know how to throw an exception, but somewhere in our code we also have to catch it and display error to the user.

try {

  person.fullName = null;

}

First we wrap the line which was causing the exception in a *try* block and any line inside this *try* block can cause an exception.

Next we catch the exception using *catch* block,

try {

  person.fullName = null;

} catch (e) {

  alert(e);

}



Let us try to catch a different type of exception, Instead of passing null now pass an empty string,

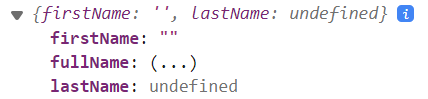
try {

  person.fullName = "";

} catch (e) {

  alert(e);

}

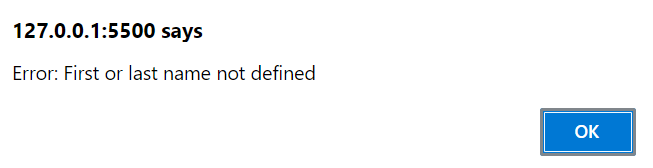


We get lastName as undefined. So we need to make sure that user is typing last and first name both.

We check the length of parts array and throw an exception in case length is not 2

    let parts = value.split(" ");

    if (parts.length !== 2) throw new Error("First or last name not defined");



Now we can see the new exception.

Note: When we throw an exception, lines after throw statement are not executed and control jumps to catch block where we catch the exception and do something with it.

const person = {

  firstName: "Himanshu",

  lastName: "Pandey",

  get fullName() {

    return `${person.firstName} ${person.lastName}`;

  },

  set fullName(value) {

    if (typeof value !== "string") throw new Error("value is not a string");

    let parts = value.split(" ");

    if (parts.length !== 2) throw new Error("First or last name not defined");

    this.firstName = parts[0];

    this.lastName = parts[1];

  },

};

try {

  person.fullName = "";

} catch (e) {

  alert(e);

}

console.log(person);

Complete code.

**Local vs Global Scope**:

Take a look at this code,

const message = "Hi";

console.log(message); //Hi

But what if we move our constant inside a block,

{

  const message = "Hi";

}

console.log(message);

we get ReferenceError: message is not defined

This is all about ***scoping***. So the scope of a variable or a constant determines where that variable or constant is accessible.

*When we declare a variable by* ***let*** *or* ***const*** *their scope is limited to the block in which they are defined*.

This block may be *function* block, *if* block, *for* loop block this scope is same for all these.

*Local vs Global*:

let color = "red"; //global variable

function start() {

  console.log(color);

}

start(); //red

But if we define the variable with same name inside this function block,

let color = "red";

function start() {

  let color = "blue";

  console.log(color);

}

start(); //blue

*Local variables/constants in a function takes precedence over global ones*. In general we should avoid declaring global variables or constants because they are available to each function globally and any function might accidently change their value.

**Let vs Var**:

So far we have been declaring variables using *let* keyword but there is another way to declare a variable using *var* keyword.

Take this code for example, where I declare the loop variable with *let* keyword.

function start() {

  for (let i = 0; i < 5; i++) {

    console.log(i);

  }

}

start(); 0 1 2 3 4

If I try to use this loop variable outside the block – scope, I will get a ReferenceError: i is not defined.

function start() {

  for (let i = 0; i < 5; i++) {

    console.log(i);

  }

  console.log(i); // ReferenceError: i is not defined

}

Let us switch this variable with *var* keyword.

function start() {

  for (var i = 0; i < 5; i++) {

    console.log(i);

  }

  console.log(i);

}

start(); 0 1 2 3 4 5

This time we did not get any error but notice that we get 5 at the end. It is because loop terminated at 4 when loop variable became 5 and that is the value we see from the second console.log()

Here is what we learned about behavior of *let* and *var* variables,

***let****,* ***const*** *defines block scoped variables*.

***var*** *is function scoped, means it will be available inside the scope of the function in which it is defined*.

Let us see this function scoped variable in more detail,

function start() {

  for (var i = 0; i < 5; i++) {

    if (true) {

      var color = "red";

    }

  }

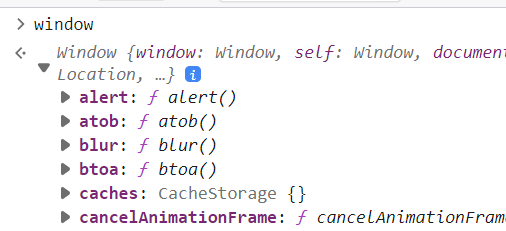
  console.log(color); //red

}

In any other programming language *color* should only be available inside *if* block. But since it is declared using ***var*** its scope is this entire *start* function.

We should not use *var* keyword with global variables because *it attaches itself to the* ***window*** *object*.

In browsers we have this window object,



which is a complex object with lots of properties and methods, very useful in creating frontend applications.

Here we defined two global variables,

var color = "red";

let age = 30;

The *var* variable has attached itself with window object but not the *let* variable.



It is really bad to attach a variable to the window object because window object is something central, there is only one instance of the window object.

Second, Let us say you are using some third party library and that library also has the same name variable. Then that variable can override your variable.

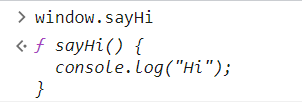
Note: When we define a function like this,

function sayHi() {

  console.log("Hi");

}

It is defined as a global function and attaches itself to window object.



It is also considered a bad practice and we can avoid it by using modules and encapsulating these functions in separate modules so they are not added to the window object.

**The “this” keyword**:

“***this***" keyword references,



Here is a simple rule of thumb,

If a function is a part of an object, we call that function a method, so if a function is a method in an object then this method references the object itself.

//method  -> obj

//function -> global (window, global)

But if a function is a regular function means not a part of any object then the function references the global object which is the window object in browsers and global in node.

Let us see the same example with respect to,

🡪 “***this***” *inside a* ***method***:

const video = {

  title: "a",

  play() {

    console.log(this); //log “***this***” on console.

  },

};

video.play(); // { title: 'a', play: [Function: play] }

Since this play function is part of this object, we can call it a method and inside this method when we log the value of *this*,

we get reference to the current *video* object.

{ title: 'a', play: [Function: play] }

Based on the same principle, if we assign a method to this object from outside like this,

const video = {

  title: "a",

  play() {

    console.log(this);

  },

};

video.stop = function () {

  console.log(this);

};

video.stop();

We get the same result, which is the reference to the current object,

{ title: 'a', play: [Function: play], stop: [Function (anonymous)] }

🡪 ***“****this****”*** *inside a* ***regular*** *function*:

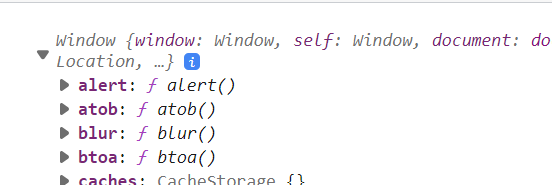
function playVideo() {

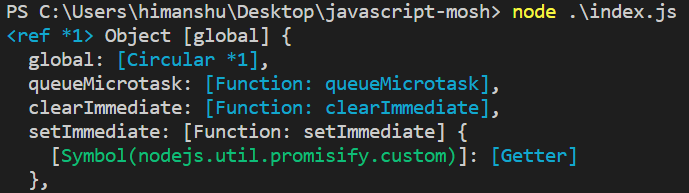
  console.log(this);

}

playVideo();

we get,

**🡨** window object in browser

🡨 global in node

🡪 ***“****this****”*** *inside a* ***constructor*** *function*:

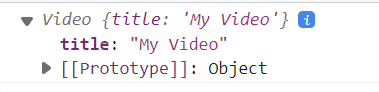
function Video(title) {

  this.title = title;

  console.log(this);

}

const v = new Video("My Video");



Here ‘*this’* refer to current object because when we use the ***new*** keyword with constructor function, it creates a new empty object and points *this* to this empty object.

Consider this example,

const video = {

  title: "a",

  tags: ["a", "b", "c"],

  showTags() {

    this.tags.forEach(function (tag) {

      console.log(tag);

    });

  },

};

video.showTags(); //a b c

We are mapping all the tags here, but what if we also want to show *title* with each tag.

const video = {

  title: "a",

  tags: ["a", "b", "c"],

  showTags() {

    this.tags.forEach(function (tag) {

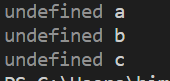
      console.log(this.title, tag);

    });

  },

};

video.showTags();

🡨 We get undefined.

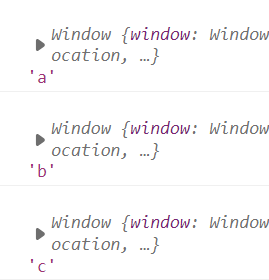
We get undefined, so let us see ‘this’ on the console.

  showTags() {

    this.tags.forEach(function (tag) {

      console.log(this, tag);

    });

 🡨 Here *this* is referencing the window object but why is that? because we are inside a method and we know that *this* inside a method refer to the object itself.

*The reason behind it*,

is because *this* is not inside method, it is inside callback function of forEach method so it is a regular function and *this* in a regular function refers to the window object.

There are multiple ways to change the value of *this* from window object to current object.

***Solution#1***:

Pass a second argument in the forEach function,

  showTags() {

    this.tags.forEach(function (tag) {

      console.log(this.title, tag); //now ***this*** refer to current object.

    }, this);

But not all functions has the second argument capability.

***Solution#2***:

Assign *this* to a constant.

  showTags() {

    const self = this;

    this.tags.forEach(function (tag) {

      console.log(self.title, tag);

    });

Not a preferred approach.

***Solution#3***:

Use *bind* method.

  showTags() {

    this.tags.forEach(

      function (tag) {

        console.log(this.title, tag);

      }.bind(this) //we bind the function manually with ***this*** using bind method

    );

  },

***Solution#4***: The cleanest and best approach is using ***arrow functions***. They *by default inherit* *this* value.

  showTags() {

    this.tags.forEach((tag) => {

      console.log(this.title, tag);

    });