JS Fundamentals Lesson 3

Mohamed Emary

August 25, 2023

1 Nullish Coalescing Operator

The nullish coalescing operator `??` provides a short syntax for selecting a first "defined" variable from the list.

The difference between `||` and `??` is that the `??` checks only for `null` and `undefined`, while the `||` operator also checks for `0`, `false` and `''` (Falsy Values).

Example:

```
let firstName = null;
let middleName = "";
let lastName = undefined;
let nickName = "Supercoder";
console.log(firstName ?? middleName ?? lastName ?? nickName ?? "Anonymous"); // "" empty
↪  string
console.log(firstName || middleName || lastName || nickName || "Anonymous"); // Supercoder
```

---

2 Topics from Odin lesson page

Important Note: I have studied this lesson topics (Fundamentals Part 3) from youtube videos not the links provided by The Odin Project

- How to define and invoke (a fancy word for run, or execute) different kinds of functions.
- How to use the return value.
- What function scope is.

In JavaScript, when defining a function, you don't have to write `let` before function parameters because function parameters are treated as variables within the function's scope. Therefore, they don't need to be declared with `let`, `const`, or `var`.

2.1 Function Rest Parameter ...

To accept any number of parameters in a function in JavaScript you can use the rest parameter `...` before the parameter name but you should take care of the following:

1. The rest parameter must be the last parameter in the function definition.
2. Only one rest parameter is allowed in a function.

Example:

```
function sum(...numbers) {
  let total = 0;
  for (let number of numbers) {
    total += number;
  }
  return total;
}
console.log(sum(1, 2, 3, 4, 5)); // 15

function sum(a, ...numbers, b) {
  // Invalid function definition
}

function sum(...numbers1, ...numbers2) {
  // Invalid function definition
}
```

2.1.1 Other Use Cases of Rest Parameter ...

- When working with array destructuring - The rest operator can be used to destructure an array and capture the remaining elements in an array.

```
const [first, ...rest] = [1, 2, 3, 4];
first // 1
rest // [2, 3, 4]
```

- When spreading an array into function arguments - The rest operator spreads an array out into individual arguments when calling a function.

```
const numbers = [1, 2, 3];
Math.max(...numbers); // 3
```

- When copying arrays - The rest operator copies all elements from one array into a new array.

```
const arr1 = [1, 2, 3];
const arr2 = [...arr1]; // arr2 = [1, 2, 3]
```

2.2 Anonymous Function

Named functions have a function name that can be called directly. Anonymous functions do not have a name bound to them they are stored in a variable and can be called using that variable.

In JavaScript Named function are ***hoisted*** to the top of their scope and can be called before they are defined. Anonymous functions can only be called after they are defined so they are not ***hoisted***.

Hoisting also happens with `var` variables which was the reason of introducing `let` and `const` variables.

Named functions show up in stack traces which can help with debugging. Anonymous functions show up as anonymous in stack traces making debugging harder.

Named functions can call themselves recursively. Anonymous functions cannot as they have no internal name to call.

Named functions can be immediately ***invoked*** by adding `()` after their declaration. Anonymous functions must be stored in a variable before being invoked.

Use named functions when:	Use anonymous functions when:
You need to recursively call the function or refer to it later in your code. Named functions make recursion and referencing easier.	The function is a throwaway helper that doesn't need recursion, reuse or referencing.
You want to attach the function to an object as a method. Named functions make it clear what the method name is.	You need to define and invoke a function inline, like in a callback. Or make a function for a specific thing like a click handler.
Readability is a concern. Named functions are easier to identify in stack traces and when reading code.	You want to define a function inside another function for encapsulation or closure.
Caching or memoization is needed. Named functions can be stored and reused easily.	The function name is not important in the current context or would not make code clearer.
	You want to use an anonymous function with higher order functions like `.filter()`, `.map()`, etc.

Anonymous function can also handle recursive calls if it was given a name but generally that doesn't happen. If you want to have recursion in you function then use named functions not anonymous ones.

Summary:

- Named functions have a name and can be called directly. Anonymous functions do not have a bound name.
- Named functions are hoisted, anonymous functions are not.
- Named functions have benefits for recursion, debugging and readability.
- Use named functions for recursion, reusability and readability.
- Use anonymous functions for throwaways that don't require naming.

Example:

```
console.log(anon_sum_hoisted(1, 2, 3, 4, 5)); // 15
function anon_sum_hoisted(...numbers) {
  let total = 0;
  for (let number of numbers) {
    total += number;
  }
  return total;
}

console.log(anon_sum_not_hoisted(1, 2, 3, 4, 5)); // Uncaught ReferenceError: Cannot access
↪  'anon_sum_not_hoisted' before initialization
let anon_sum_not_hoisted = function (...numbers) {
  let total = 0;
  for (let number of numbers) {
    total += number;
  }
  return total;
};

anon_sum_not_hoisted(1, 2, 3, 4, 5);  // 15
                                      // function is accessible after definition via variable
                                      ↪  name
```

The second function definition `anon_sum_not_hoisted` is not hoisted to the top of the file and it's not accessible before the definition line.

Notice that since `anon_sum_not_hoisted` is an anonymous function the inline function definition doesn't have a name you can also have a name in that function but it will be only accessible inside the function.

The anonymous function is declared and initialized when their definition is evaluated at parse time before runtime, and it's available for **invocation (function calling)** right away.

The JavaScript engine compiles the source code before execution. This parses and evaluates function definitions.

Example:

```
// Anonymous function declared and initialized here
const myFunc = function() {
  console.log('Hi!');
};

// Runtime starts here
myFunc(); // 'Hi!'
```

Anonymous function can has a name but it will be only accessible inside the function and if you try to access it outside the function you will get a `ReferenceError`.

2.3 Nesting Functions in JavaScript

Functions in JavaScript can be nested inside other functions. This is useful for encapsulation. Nested functions are only accessible inside the function they are nested in.

When nesting functions you can even return the nested function from the outer function. This is called a ***closure***. The nested function will still have access to the outer function's variables even after the outer function has returned.

Example 1:

```
function sayMessage(fName, lName) {
  let message = `Hello`;
  // Nested Function
  function concatMsg() {
    message = `${message} ${fName} ${lName}`;
  }
  concatMsg();
  return message;
}

console.log(sayMessage("Osama", "Mohamed"));
```

Example 2:

```
function sayMessage(fName, lName) {
  let message = `Hello`;
  // Nested Function
  function concatMsg() {
    return `${message} ${fName} ${lName}`;
  }
  return concatMsg();
}

console.log(sayMessage("Osama", "Mohamed"));
```

Example 3:

```
function sayMessage(fName, lName) {
  let message = `Hello`;
  // Nested Function
  function concatMsg() {
    function getFullName() {
      return `${fName} ${lName}`;
    }
    return `${message} ${getFullName()}`;
  }
  return concatMsg();
}

console.log(sayMessage("Osama", "Mohamed"));
```

Example 4:

```
function get_info(first_name, last_name,
↪  birth_year) {
  function full_name() {
    return `${first_name} ${last_name}`;
  }

  function age() {
    return 2023 - birth_year;
  }

  function info() {
    let info = `Your name is ${full_name()}
    ↪  and you are ${age()} yo`;
    return info;
  }

  return info();
}

console.log(get_info("Mohamed", "Ahmed",
↪  2010));
```

2.4 Arrow Function

Arrow functions is JavaScript are a shorthand for writing functions. They are written using the `=>` syntax. They are anonymous functions and must be stored in a variable to be used.

Since they are anonymous functions they are given no name and you also don't need to use the `function` keyword.

Example:

```
// get the square root of the sum of two numbers
let sum_root = (num1, num2) => {
  sum = num1 + num2
  return Math.sqrt(sum)
}
console.log(sum_root(10, 6));  // 4
```

In arrow functions you can remove the `( )` if you have only one parameter, you can also remove the `{ }` and the `return` keyword if you have only one statement. Also if you have no parameters at all you can replace the empty parenthesis `( )` with underscore `_`

```
// Example of removing `return` and `{ }`
let get_sum = (num1, num2) => num1 + num2
console.log(get_sum(30, 20)); // 50

// Example of removing `( )`, `{ }`, and `return`
let plus30 = num => num + 30
console.log(plus30(30));  // 60

// Example of replacing `( )` with `_`
let get30 = _ => 30
console.log(get30(30)); // 30
```

2.5 Variable Scope

The scope of a variable is the region of your program in which it is defined. JavaScript variable scope can be either local or global. **Global scope** variables are accessible from anywhere in your code. **Local scope** variables are only accessible from within the function or block that they are defined.

When you declare a variable outside of any function, it is called a global variable, because it is available to any other code in the current document. When you declare a variable within a function, it is called a local variable, because it is available only within that function.

Example 1:

```
var a = 1;
let b = 2;

function scope() {
  console.log(`a = ${a}`);  // undefined
  console.log(`b = ${b}`);  // Cannot access
  ↪  'b' before initialization
  var a = 10;
  let b = 20;
}

scope();

console.log(a); // 1
console.log(b); // 2
```

Example 2:

```
var c = 1;
let d = 2;

function scope() {
  var c = 10;
  let d = 20;
  console.log(`c = ${c}`);  // 10
  console.log(`d = ${d}`);  // 20
}

scope();

console.log(c); // 1
console.log(d); // 2
```

In both examples we see that the scope of variables defined inside a function is only the function.

In *example 1* if you comment the line `let b = 2;` inside the function you will get `b = 2`, also if you comment the line `var a = 10;` inside the function you will get `a = 1`.

Functions have their own scope in JavaScript. Variables defined inside a function are not accessible from outside the function but variables defined outside a function are accessible from inside the function.

Loops, if statements, and other blocks of code do not have their own scope. Variables defined inside a block of code are accessible from outside the block. This behavior depend on whether you used `var`, `let`, or `const` to define the variable/constant because `var` is function scoped but not block scoped but `let` and `const` are function and block scoped.

Example 1:

```
let a = 5;
var b = 3;

if (true) {
  // This is a different variable than the
  // one defined before and if it wasn't you
  // could have got an error because you
  // can't redeclare a variable using `let`
  let a = 50;
  // This overwrites the variable b
  var b = 30;
  console.log(a); // 50
  console.log(b); // 30
}
console.log(a); // 5
console.log(b); // 30 not 3
```

Example 2:

```
// notice here a uses `var` and b uses `let`
var a = 5;
let b = 3;

if (true) {
  console.log(a); // 5
  console.log(b); // Cannot access 'b' before
  ↪  initialization
  // This overwrites the variable a
  var a = 50;
  // This is a different variable than the one
  ↪  defined before
  let b = 30;
}
console.log(a); // 5
console.log(b); // 30 not 3
```

The `Cannot access 'b' before initialization` error happens because the `console.log()` has found a variable `b` in the block scope but it's not initialized yet so it throws an error. This part of the code is called the ***Temporal Dead Zone (TDZ)***.

Temporal Dead Zone (TDZ) is the area of block where a variable is inaccessible until the moment the computer completely initializes it with a value.

The last scope we are going to discuss is the ***Lexical Scope***. Since we can nest functions in JavaScript, functions can be defined inside other functions. The inner function can access the variables of the outer function but the outer function cannot access the variables of the inner function this is the ***Lexical Scope***.

Example 1:

```
function outer1(){
  let a = 10;
  function inner(){
    console.log(a); // 10
  }
  inner();
}

outer1()
```

Example 2:

```
function outer2(){
  let a = 10;
  function inner(){
    let a = 100;
    console.log(a); // 100
  }
  console.log(a); // 10
  inner();
}

outer2()
```

Example 3:

```
function outer() {
  let a = 10;
  function inner() {
    let a = 100;
    let b = 20;
    console.log(a); // 100
  }
  inner();
  console.log(b); // Uncaught ReferenceError: b is not defined
}

outer();
```

3 Some Notes From Articles

- **Read this JavaScript.info article about function expressions the whole article is important and has many valuable information.**
- Also Search for a YouTube video to explain **callback functions**.
- The Last Thing is to read this article about **call stack** and **execution contexts** (Global execution context & function execution contexts) in JavaScript.

A call stack is a way for the JavaScript engine to keep track of its place in code that calls multiple functions. It has the information on what function is currently being run and what functions are invoked from within that function...

Whenever a function is called, the JavaScript engine creates a function execution context for the function, pushes it on top of the call stack, and starts executing the function.

When the JavaScript engine executes this script, it places the global execution context denoted by `main()` or `global()` function on the call stack.

The call stack has a fixed size, depending on the implementation of the host environment, either the web browser or *Node.js*. If the number of execution contexts exceeds the size of the stack, a stack overflow error will occur.

Asynchronous JavaScript

JavaScript is a single-threaded programming language. This means that the **JavaScript engine has only one call stack**. Therefore, **it only can do one thing at a time**.

When executing a script, the **JavaScript engine executes code from top to bottom**, line by line. In other words, it is **synchronous**.

Asynchronous means the **JavaScript engine can execute other tasks while waiting for another task to be completed**. For example, the JavaScript engine can:

1. Request for data from a remote server.
2. Display a spinner
3. When the data is available, display it on the webpage.

To do this, the JavaScript engine uses an **event loop**.

---

some of the code you are calling when you invoke a built in browser function couldn't be written in JavaScript - many of these functions are calling parts of the background browser code, which is written largely in low-level system languages like C++, not web languages like JavaScript.

some built-in browser functions are not part of the core JavaScript language - some are defined as part of browser APIs, which build on top of the default language to provide even more functionality.

Functions that are part of objects are called **methods**.

Some functions require parameters to be specified when you are invoking them - these are values that need to be included inside the function parentheses, which it needs to do its job properly. Function **Parameters** are sometimes called **arguments**, **properties**, or even *attributes*.

Take a look at this exercise.

Unlike python JavaScript doesn't return multiple values from a function but you can return a single array of multiple values.

Think about the idea of creating a function library. As you go further into your programming career, you'll start doing the same kinds of things over and over again. It is a good idea to create your own library of utility functions to do these sorts of things. You can copy them over to new code, or even just apply them to HTML pages wherever you need them.

A **parameter** is the variable listed inside the parentheses in the function declaration (it's a declaration time term). An **argument** is the value that is passed to the function when it is called (it's a call time term).

We declare functions listing their parameters, then call them passing arguments.

Function **parameters** are the items listed between the parentheses **in the function declaration**. Function **arguments** are the **actual values** we decide to pass to the function.

A function parameter is just a **placeholder** for some future value

In this code We're passing in a function call `favoriteAnimal('Goat')` as an argument in a different function call `log()`

```
function favoriteAnimal(animal) {
  return animal + " is my favorite animal!"
}

console.log(favoriteAnimal('Goat'))
```

If a same-named variable is declared inside the function then it shadows the outer one. It's a good practice to minimize the use of global variables. Modern code has few or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

When the function is called in lines `*` and `**`, the given values are copied to local variables `from` and `text`. Then the function uses them.

Here's one more example: we have a variable `from` and pass it to the function. Please note: the function changes `from`, but the change is not seen outside, because a function always gets a copy of the value:

```
function showMessage(from, text) {
  from = '*' + from + '*'; // make "from"
  ↪  look nicer
  alert( from + ': ' + text );
}

let from = "Ann";
showMessage(from, "Hello"); // *Ann*: Hello
// the value of "from" is the same, the
↪  function modified a local copy
alert( from ); // Ann
```

When using default function parameters you can have complex expression that are the default value of the parameter, for example it can be a function call.

```
function showMessage(from, text =
↪  anotherFunction()) {
  // anotherFunction() only executed if no
  ↪  text given
  // its result becomes the value of text
}
```

If a function is called, but an argument is not provided, then the corresponding value becomes `undefined`.

For instance, the aforementioned function `showMessage(from, text)` can be called with a single argument: `showMessage("Ann");`

That's not an error. Such a call would output `"*Ann*: undefined"`. As the value for text isn't passed, it becomes undefined.

We can specify the so-called "default" (to use if omitted) value for a parameter in the function declaration, using `=`

```
function showMessage(from, text = "no text
↪  given") {
  alert( from + ": " + text );
}

showMessage("Ann"); // Ann: no text given
```

Default parameters in old JavaScript code

In older JS code people used to see a trick like this:

```
function showMessage(from, text) {
  if (text === undefined) {
    text = 'no text given';
  }

  alert( from + ": " + text );
}
```

They also used the or `||` operator

```
function showMessage(from, text) {
  // If the value of text is falsy, assign
  ↪  the default value
  // this assumes that text == "" is the same
  ↪  as no text at all
  text = text || 'no text given';
  ...
}
```

You can also use the *nullish coalescing operator* `??` operator like that `alert(count ?? "unknown");`.

Function return value can also be another function call.

If a function does not return a value, it is the same as if it returns `undefined`

```
function doNothing() { /* empty */ }
alert( doNothing() === undefined ); // true
```

Never add a newline between `return` and the value because JavaScript assumes a semicolon `;` after `return` So, it effectively becomes an empty `return`.

you can use this workaround

```
return (
  some + long + expression
  + or +
  whatever * f(a) + f(b)
  )
```

Functions are actions. So their name is usually a verb. It should be brief, as accurate as possible and describe what the function does, so that someone reading the code gets an indication of what the function does.

It is a widespread practice to start a function with a verbal prefix which vaguely describes the action. There must be an agreement within the team on the meaning of the prefixes.

For instance, functions that start with `show` usually show something.

Function starting with...

- `get...` – return a value,
- `calc...` – calculate something,
- `create...` – create something,
- `check...` – check something and return a boolean, etc.

```
showMessage(..) // shows a message
getAge(..)  // returns the age (gets it somehow)
calcSum(..) // calculates a sum and returns the result
createForm(..)  // creates a form (and usually returns it)
checkPermission(..) // checks a permission, returns true/false
```

With prefixes in place, a glance at a function name gives an understanding what kind of work it does and what kind of value it returns.

One function – one action

A function should do exactly what is suggested by its name, no more.

Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two).

A few examples of breaking this rule:

`checkPermission` – would be bad if it displays the `access granted/denied` message (should only perform the check and return the result).

Functions that are used very often sometimes have **ultrashort names**. For example, the jQuery framework defines a function with `$`. The Lodash library has its core function named `_`.

When we define a function and assign it to a variable it is called a **Function Expression**.

In JavaScript you can assign a function to a variable and use that variable to call the function. Since you have that ability you can also assign one function to multiple variables and call it from any of them.