JS Fundamentals Lesson 3

Mohamed Emary

August 23, 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 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
```