Route Academy

JavaScript Basics: Week 2

A Guide for Beginners

Prepared for Students at Route Academy

May 2025

Contents

1 Welcome to Week 2 of JavaScript!

Hello, Route Academy stars! Last week, we learned how to store things in variables, make decisions with conditionals, and repeat tasks with loops. This week, we're going to make our code even more exciting! We'll start with **functions** to create reusable recipes, learn where variables live with **scope**, understand JavaScript's magic trick called **hoisting**, organize information with **objects** and **arrays**, and explore **array methods** to manage lists. Let's dive in with some fun examples!

2 Functions: Your Code's Superpowers

A **function** is like a little robot you create to do a specific job for you. You tell it what to do, and then you can call it whenever you need that job done. It's like having a magic button you can press to make things happen!

2.1 Creating a Function (Declaration)

The classic way to write a function is called a *declaration*, using the `function` keyword followed by a name.

```
function sayHello() {
    console.log("Hello, Route Academy!");
}
sayHello(); // Output: Hello, Route Academy!
```

This function is like a recipe book that's always ready to use. You can call it as many times as you want:

```
sayHello(); // Output: Hello, Route Academy!
sayHello(); // Output: Hello, Route Academy!
```

2.2 Function Expression

Another way to write a function is called an *expression*, where you store the function in a variable, like putting the recipe in a box.

```
var sayHello = function() {
    console.log("Hello, Route Academy!");
};
sayHello(); // Output: Hello, Route Academy!
```

The function doesn't have a name on its own—it's stored in the variable `sayHello`.

2.3 What's the Difference?

The big difference between declaration and expression is *hoisting* (we'll talk about that soon!). A *declaration* can be called before it's written in the code, but an *expression* cannot.

```
sayHello(); // Works because of hoisting
function sayHello() {
    console.log("Hello!");
}

sayHi(); // Error: sayHi is not a function
var sayHi = function() {
    console.log("Hi!");
};
```

2.4 Returning Values from Functions

Functions can give you something back using the `return` keyword. Once a function hits `return`, it stops and gives back the value.

```
function addNumbers(x, y) {
    var result = x + y;
    return result; // Gives back the result and stops
    return 10; // This 'wont run
    console.log("This 'wont run!");
}
console.log(addNumbers(10, 20)); // Output: 30
```

Here, `return result` gives back 30, and the function stops, so the lines after it don't run.

2.5 Example: A Candy Calculator

Let's create a function to calculate how many candies you can buy!

```
function buyCandies(money, pricePerCandy) {
    var candies = money / pricePerCandy;
    return candies;
}
var candiesICanBuy = buyCandies(10, 2); // 10 dollars, 2 dollars
   per candy
console.log("I can buy " + candiesICanBuy + " candies!"); //
   Output: I can buy 5 candies!
```

3 Scope: Where Variables Live

Scope is like the "home" of a variable—it tells you where a variable can be used. There are two main homes: *global scope* (outside any function) and *local scope* (inside a function).

3.1 Global Scope

Variables created outside of any function live in the *global scope*, which means they can be used anywhere in your code.

```
var globalToy = "Teddy Bear"; // This variable lives globally
function playWithToy() {
    console.log("Playing with my " + globalToy);
}
playWithToy(); // Output: Playing with my Teddy Bear
console.log(globalToy); // Output: Teddy Bear
```

Since `globalToy` is global, we can use it inside the function and outside it.

3.2 Local Scope

Variables created inside a function live in the *local scope*, which means they can only be used inside that function.

```
function eatSnack() {
    var snack = "Cookies"; // This variable lives only inside the
       function
    console.log("Eating some " + snack);
}
eatSnack(); // Output: Eating some Cookies
console.log(snack); // Error: snack is not defined (it only exists
   inside the function)
```

Here, `snack` is local to the function, so we can't use it outside.

3.3 Example: A Secret Treasure Chest

Let's imagine you have a treasure chest that only opens inside a special room (a function).

```
var roomKey = "Golden Key"; // Global variable (everyone can see it)
function openTreasureChest() {
    var treasure = "Gold Coins"; // Local variable (only exists
       inside the function)
    console.log("Using the " + roomKey + " to find " + treasure);
}
openTreasureChest(); // Output: Using the Golden Key to find Gold
   Coins
console.log(treasure); // Error: treasure is not defined ('its
   locked inside the function)
```

4 Hoisting: JavaScript's Magic Trick

Hoisting is like JavaScript preparing your code before running it. It moves all *function declarations* and *variable declarations* (but not their values) to the top of the code.

4.1 Hoisting with Variables

When you declare a variable with `var`, JavaScript moves the declaration to the top, but the value stays where it is.

```
console.log("Hello");
console.log(userName); // Output: undefined
var userName = "Ahmed";
var userAge = 30;
console.log("Hello");
```

JavaScript sees this as:

```
var userName; // Declaration is hoisted
var userAge; // Declaration is hoisted
console.log("Hello");
console.log(userName); // undefined (no value yet)
userName = "Ahmed";
userAge = 30;
console.log("Hello");
```

4.2 Hoisting with Function Declarations

Function declarations are fully hoisted, meaning you can call them before they're written.

```
foo(); // Output: 8
function foo() {
    function bar() { return 3; }
    return bar();
    function bar() { return 8; }
}
```

JavaScript hoists both `foo` and `bar` functions:

```
function foo() {
    function bar() { return 3; }
    function bar() { return 8; } // The second bar overwrites the
       first
    return bar(); // So bar() returns 8
}
foo(); // Output: 8
```

4.3 Hoisting with Function Expressions

Function expressions are not fully hoisted—only the variable declaration is hoisted, not the function itself.

```
foo(); // Error: foo is not a function
var foo = function() {
    var bar = function() { return 3; }
    return bar();
    var bar = function() { return 8; }
};
```

JavaScript sees this as:

```
var foo; // Declaration is hoisted
foo(); // Error: foo is undefined, not a function
foo = function() {
    var bar = function() { return 3; }
    return bar();
    var bar = function() { return 8; }
};
```

4.4 Mixed Hoisting Example

Let's mix declarations and expressions:

```
foo(); // Output: 3
function foo() {
    return bar();
    function bar() { return 3; }
    var bar = function() { return 8; }
}
```

JavaScript hoists the declaration first:

```
function foo() {
    function bar() { return 3; } // Hoisted declaration
    return bar(); // Uses the declared bar, returns 3
    var bar; // Declaration hoisted
    bar = function() { return 8; }; // Assignment stays here
}
```

4.5 Example: A Surprise Party

Let's see hoisting in action with a party scenario:

```
planParty(); // Output: 'Lets have a party!
function planParty() {
    console.log("'Lets have a party!");
}

surprise(); // Error: surprise is not a function
var surprise = function() {
    console.log("Surprise!");
};
```

The declaration `planParty` is hoisted, so it works, but `surprise` is an expression, so it doesn't.

5 Objects: Your Super Organizer

An **object** is like a magical backpack that can hold many things at once—names, numbers, and even actions! Instead of having separate variables for everything, you can keep related things together in one place.

5.1 Creating an Object

To make an object, use curly braces `{}` and put your items inside as *key-value pairs*, separated by commas. The *key* is like a label, and the *value* is what's inside that label.

```
var car = {
    name: "Nissan",
    model: 200,
    color: "black"
};
console.log(car); // Output: {name: "Nissan", model: 200, color:
   "black"}
```

This `car` object is like a backpack with three pockets: one labeled `name` with "Nissan" inside, one labeled `model` with 200, and one labeled `color` with "black".

5.2 Adding Actions to Objects (Methods)

Objects can also hold actions, called *methods*, which are functions inside the object. Let's add some actions to our car!

```
var car = {
    name: "Nissan",
    model: 200,
    color: "black",
    start: function() {
        console.log("Start the car!");
    },
    stop: function() {
        console.log("Stop the car!");
    }
};
car.start(); // Output: Start the car!
car.stop(); // Output: Stop the car!
```

Here, `start` and `stop` are methods. When we call `car.start()`, it's like pressing a button on the car to start it.

5.3 Accessing Object Properties

To look inside an object's pockets, you can use three ways:

- **Dot notation**: `objectName.key` (like opening a pocket with a label)
- **Bracket notation**: `objectName["key"]` (like looking up the label in a list)
- **Calling a method**: `objectName.methodName()` (to run an action)

```
var car = {
    name: "Nissan",
    model: 200,
    color: "black",
    start: function() {
```

```
        console.log("Start the car!");
    }
};
console.log(car.name); // Output: Nissan (dot notation)
console.log(car["color"]); // Output: black (bracket notation)
car.start(); // Output: Start the car! (calling a method)
```

5.4 Nested Objects: Backpacks Inside Backpacks

Objects can hold other objects, like a small backpack inside a bigger one! Let's create a family object with nested information.

```
var user = {
    fullName: "Ali",
    age: 50,
    eyeColor: "blue",
    gender: "male",
    eat: function() {
        console.log("Yum, 'Im eating!");
    },
    wife: {
        name: "Aya",
        age: 30,
        son: {
            name: "Ahmed Ali",
            age: 15,
            gender: "male"
        }
    }
};
```

This `user` object has a `wife` object inside it, and the `wife` object has a `son` object inside it! Let's access the information:

```
console.log(user); // Output: (the entire user object)
console.log(user.fullName); // Output: Ali
console.log(user.eat()); // Output: Yum, 'Im eating!
console.log(user.wife); // Output: (the entire wife object)
console.log(user.wife.name); // Output: Aya
console.log(user.wife.son); // Output: (the entire son object)
console.log(user.wife.son.name); // Output: Ahmed Ali
```

We use dot notation to dig deeper into the nested objects, like opening smaller backpacks inside the big one.

5.5 Built-in Objects: JavaScript's Helpers

JavaScript comes with some special objects that help you do amazing things! Here are a few:

- `window`: The big boss of your webpage—it controls everything, like opening alerts (`window.alert`).

- `document`: Helps you talk to your webpage, like changing text (`document.getElementById`).
- `console`: Your best friend for printing messages (`console.log`).
- `screen`: Tells you about the user's screen, like its size (`screen.width`).
- `Array`: A special tool to create lists (we'll learn more about arrays next!).

We've already used `console.log` and `window.alert`—they're part of these built-in objects!

5.6 Example: Your Dream Robot

Let's create an object for a dream robot that helps you with tasks!

```
var robot = {
    name: "HelperBot",
    color: "Silver",
    tasks: {
        clean: function() {
            console.log("Cleaning the room!");
        },
        cook: function() {
            console.log("Cooking a delicious meal!");
        }
    }
};
console.log(robot.name + " is " + robot.color); // Output:
   HelperBot is Silver
robot.tasks.clean(); // Output: Cleaning the room!
robot.tasks.cook(); // Output: Cooking a delicious meal!
```

6 Arrays: Your Magic List

An **array** is like a magic list where you can store many things in one place, like a row of boxes labeled with numbers starting from 0.

6.1 Why Use Arrays?

Imagine you have a list of cars, and you store them like this:

```
var car1 = "Saab";
var car2 = "Volvo";
var car3 = "BMW";
```

What if you have 300 cars? It would be hard to manage! Arrays solve this by letting you store everything in one list, and you can loop through them easily.

6.2 Creating an Array

To make an array, use square brackets `[]` and put your items inside, separated by commas.

```
var friends = ["Ahmed", "Ali", "Mohamed", "Abdo"];
console.log(friends); // Output: ["Ahmed", "Ali", "Mohamed", "Abdo"]
```

This array is a list of friends. The first friend, "Ahmed", is in box 0, "Ali" is in box 1, and so on.

6.3 Accessing Array Elements

To look inside a box, use the array name and the box number (called the *index*) in square brackets. Remember: the first box is 0!

```
var friends = ["Ahmed", "Ali", "Mohamed", "Abdo"];
console.log(friends[0]); // Output: Ahmed (first box)
console.log(friends[1]); // Output: Ali (second box)
console.log(friends[2]); // Output: Mohamed (third box)
console.log(friends[3]); // Output: Abdo (fourth box)
```

6.4 The `length` Property

The `length` property tells you how many boxes are in your array.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
var numberOfFruits = fruits.length;
console.log(numberOfFruits); // Output: 4
```

There are 4 fruits, so the length is 4. The last fruit's index is `length - 1`, which is 3.

6.5 Looping Through an Array

You can use a `for` loop to visit each box in the array, like walking down a row of boxes and opening each one.

```
var fruits = ["Banana", "Orange", "Apple", "Mango"];
for (var i = 0; i < fruits.length; i++) {
    console.log(fruits[i]);
}
// Output:
// Banana
// Orange
// Apple
// Mango
```

The loop starts at `i = 0`, keeps going until `i` reaches the length (4), and adds 1 to `i` each time.

7 Array Methods: Magic Tricks for Lists

Arrays have special *methods* that let you add, remove, or change items easily.

7.1 Adding and Removing Items

- `push`: Adds an item to the end of the list.
- `pop`: Removes the last item from the list.
- `shift`: Removes the first item from the list.
- `unshift`: Adds an item to the beginning of the list.

```
var colors = ["Blue", "Yellow", "Gray"];
colors.push("Black"); // Add Black to the end
console.log(colors); // Output: ["Blue", "Yellow", "Gray", "Black"]
colors.pop(); // Remove the last item (Black)
console.log(colors); // Output: ["Blue", "Yellow", "Gray"]
colors.shift(); // Remove the first item (Blue)
console.log(colors); // Output: ["Yellow", "Gray"]
colors.unshift("Red"); // Add Red to the beginning
console.log(colors); // Output: ["Red", "Yellow", "Gray"]
```

7.2 Slicing and Joining Arrays

- `slice(start, end)`: Copies a part of the array into a new array.
- `concat`: Joins two arrays together.

```
var colors = ["Red", "Yellow", "Gray", "Black", "Tomato"];
var someColors = colors.slice(1, 3); // Copy from index 1 to (3-1)
console.log(someColors); // Output: ["Yellow", "Gray"]
var moreColors = colors.concat(["Purple", "Pink"]); // Join arrays
console.log(moreColors); // Output: ["Red", "Yellow", "Gray",
   "Black", "Tomato", "Purple", "Pink"]
```

7.3 Sorting and Reversing

- `sort`: Sorts the array in alphabetical order.
- `reverse`: Reverses the order of the array.

```
var colors = ["Yellow", "Red", "Gray"];
colors.sort(); // Sort alphabetically
console.log(colors); // Output: ["Gray", "Red", "Yellow"]
colors.reverse(); // Reverse the order
console.log(colors); // Output: ["Yellow", "Red", "Gray"]
```

7.4 Example: Your Superhero Team

Let's use array methods to manage your superhero team!

```
var heroes = ["Spider-Man", "Iron Man", "Hulk"];
console.log("My team: " + heroes); // Output: My team:
    Spider-Man,Iron Man,Hulk
heroes.push("Captain America"); // Add a new hero
console.log("New team: " + heroes); // Output: New team:
    Spider-Man,Iron Man,Hulk,Captain America
heroes.sort(); // Sort the team
console.log("Sorted team: " + heroes); // Output: Sorted team:
    Captain America,Hulk,Iron Man,Spider-Man
```

8 Common Interview Questions for Week 2

Let's get ready for some interview questions to test your new skills!

8.1 Question 1: What's the Difference Between Declaration and Expression?

Answer: A *function declaration* uses `function name() {}` and is fully hoisted, so you can call it before it's written. A *function expression* is stored in a variable (like `var name = function() {}`) and isn't hoisted, so you can't call it before it's defined.

8.2 Question 2: What Does This Code Output?

```
function foo() {
    var x = 10;
    var y = 20;
    var result = x + y;
    return result;
    return 10;
    console.log("Hello");
}
console.log(foo()); // Output: 30
```

Answer: The function returns `result` (30), and stops there. The lines after `return result` (like `return 10` and `console.log`) don't run.

8.3 Question 3: What Happens Here with Hoisting?

```
console.log(foo());
var foo = function() {
    return bar();
    function bar() { return 3; }
    var bar = function() { return 8; }
};
```

Answer: This throws an error: `foo is not a function`. JavaScript hoists the declaration `var foo`, but not the function expression, so at the time of `console.log(foo())`, `foo` is `undefined`.

8.4 Question 4: What's the Output of This Array Code?

```
var fruits = ["Banana", "Orange", "Apple"];
fruits.push("Mango");
fruits.shift();
console.log(fruits); // Output: ["Orange", "Apple", "Mango"]
```

Answer: `push("Mango")` adds "Mango" to the end, making `["Banana", "Orange", "Apple", "Mango"]`. Then `shift()` removes the first item ("Banana"), leaving `["Orange", "Apple", "Mango"]`.

8.5 Question 5: How Do You Access This Nested Object Property?

```
var user = {
    wife: {
        son: {
            name: "Ahmed Ali"
        }
    }
};
```

Answer: Use dot notation to dig into the nested objects:

```
console.log(user.wife.son.name); // Output: Ahmed Ali
```