Foundational modern JavaScript syntax and practical examples

3.1 Variable Declarations: `let` and `const` vs `var`

Modern JavaScript introduced two new ways to declare variables: **`let`** and **`const`**, which differ significantly from the old **`var`**. In other programming languages, you might be used to block-scoped variables, where a variable exists only within the `{ ... }` block it's defined in. In JavaScript, `let` and `const` provide this familiar block scope. By contrast, `var` has function scope or global scope, meaning a `var` declared inside a function is visible throughout that entire function (no matter the block), and a `var` outside any function becomes a global variable. This difference in scope can lead to unexpected behavior if `var` is used in loops or conditional blocks, as it does **not** confine itself to the block. Additionally, `let` and `const` do not allow redeclaring the same variable in the same scope, whereas `var` would silently allow redefinitions. Using `const` further signals that the variable is meant to remain constant (its value cannot be reassigned after initialization).

Scope and Hoisting: Variables declared with `let` and `const` are in a "temporal dead zone" until their definition is evaluated, meaning you cannot use them *before* they appear in the code. In contrast, `var` declarations are hoisted (conceptually "lifted" to the top of their scope), defaulting to `undefined` if used before assignment. In practice, this means you should **prefer** `let`/`const` for predictable, block-scoped behavior, and reserve `var` only for legacy code. Below is a demonstration of scope differences between `var` and `let`:

```
// Using var vs let inside a block
var x = 10;
if (true) {
    var x = 20;        // var does not respect block, this reassigns the global x
    let y = 30;        // y exists only within this if-block
    console.log("inside if, x:", x); // -> inside if, x: 20
    console.log("inside if, y:", y); // -> inside if, y: 30
}
console.log("outside if, x:", x);     // -> outside if, x: 20

// y is not accessible here; using typeof to check prevents a ReferenceError
if (typeof y !== 'undefined') {
    console.log("outside if, y:", y);
} else {
    console.log("outside if, y is not defined"); // -> outside if, y is not defined
}
```

In the code above, the `var x` defined inside the `if` **overwrote** the `x` outside, because `var` has function (or global) scope. The `let y` inside the block, however, ceased to exist after the block ended. Attempting to use `y` outside its block would cause a *ReferenceError*. We used `typeof y` as a safe way to check `y`'s existence without throwing an error.

Best Practices: Use `let` for variables that will change and `const` for values that should not change. This makes your code more robust by preventing accidental reassignments. `var` is now considered outdated in modern JavaScript development due to its scoping quirks and is generally avoided.

Exercise: Try rewriting a small program that uses multiple `var` declarations by replacing them with `let` and `const`. Observe how the scope behaviour changes – for example, declare a variable inside a loop with `let` and print it outside the loop (it should be undefined or cause an error), then do the same with `var` (it will unexpectedly still exist).

3.2 Arrow Functions (=>) vs Regular Functions

JavaScript's **arrow functions** (=>) provide a concise syntax for writing functions, introduced in ES6 (2015). If you have used lambda expressions in other languages (like Python's lambda or C#'s lambda functions), arrow functions are conceptually similar. They allow you to define anonymous functions more succinctly, often inlined where they are needed (for example, as callbacks). Beyond syntax, arrow functions come with some semantic differences from traditional **function** expressions/declarations: - **Syntax:** Arrow functions omit the `function` keyword. For instance, `(a, b) => a + b` defines a function that returns the sum of `a` and `b`. For single-parameter functions, you can even omit the parentheses (e.g. `x => x * 2`). If the function body is a single expression, the arrow function implicitly returns that value (no need for `return` or braces). - **`this` Binding:** Unlike regular functions, arrow functions do **not** have their own `this`. They capture the `this` value of the surrounding context (lexical `this`). This is useful when you need to preserve the context inside callback functions. Regular functions, on the other hand, have their `this` determined by how they are called (or default to global `this` in non-strict mode). - **No `arguments` object:** Arrow functions also do not have the built-in `arguments` object. If you need to handle a variable number of parameters inside an arrow function, you would use rest syntax (`...args`, discussed later) instead of `arguments`. - **Not constructible:** You cannot use an arrow function as a constructor (i.e. you cannot call it with `new`), whereas regular functions can be constructors. Arrow functions are meant for short callback or functional-style uses rather than object blueprints.

Despite these differences, in many everyday cases an arrow function can replace a regular function expression. They shine in situations like array methods or event handlers where writing a full function would be verbose. Below is a comparison using a regular anonymous function versus an arrow function to achieve the same result:

```
const numbers = [1, 2, 3];
const doubled1 = numbers.map(function(n) {
    return n * 2;
});
const doubled2 = numbers.map(n => n * 2);

console.log("Normal function result:", doubled1); // -> Normal function result: [ 2,
4, 6 ]
console.log("Arrow function result:", doubled2);  // -> Arrow function result: [ 2, 4,
6 ]
```

In this example, both `doubled1` and `doubled2` end up with the array `[2, 4, 6]`. The arrow function version is shorter and clearer. The arrow function `n => n * 2` implicitly returns `n * 2` for each element. A regular function requires more boilerplate (`function` keyword and an explicit `return`). For simple transformations, arrows make the code succinct.

Another subtle difference shown here is that arrow functions, being expressions, are not hoisted like function declarations. You must define an arrow function before you use it (just as with any variable defined via `let`/`const`). Regular function *declarations* (using the `function myFunc() {}` syntax at top level) are hoisted, meaning you can call them even if the definition appears later in the code. This hoisting does **not** apply to arrow functions assigned to variables.

Exercise: Take a simple function in JavaScript (for example, a function that calculates the square of a number or concatenates two strings) and rewrite it as an arrow function. Try also writing a method inside an object using a regular function and an arrow function to observe how the value of `this` differs in each case when the method is called.

3.3 Rest and spread syntax and destructuring

Modern JavaScript provides the **spread operator** and **rest parameter** (both use the `...` syntax) to handle collections of data in a flexible way. These features often go hand-in-hand with **destructuring assignment**, which lets you unpack arrays or objects easily. If you are coming from languages like Python, you can think of `...` in JavaScript as somewhat analogous to Python's `*args` (for rest parameters) and the unpacking operator for spreading arrays, and destructuring is akin to tuple unpacking.

> **Rest Parameter (`...`):** The rest syntax allows a function to accept an *indefinite number of arguments* as an array. You use it in a function definition like `function myFunc(...args) { }`. This collects all actual arguments into an array named `args`. It's extremely useful for variadic functions (functions that can take a variable number of inputs) without resorting to the old `arguments` object. Rest syntax can also be used in array or object destructuring to collect the "rest" of the values. For example, `[first, ...remaining] = [1, 2, 3, 4]` assigns `first = 1` and `remaining = [2, 3, 4]`.
>
> **Spread Operator (`...`):** In contrast, spread syntax *expands* an array or object into individual elements. You can use it to combine arrays or to pass array elements as separate arguments to a function. For arrays: `let arr3 = [...arr1, ...arr2]` concatenates two arrays. For objects: `let clone = {...originalObj}` creates a shallow copy of an object. Spread is essentially the inverse of rest – where rest gathers multiple elements into one, spread takes one iterable (like an array or string) and expands it into multiple elements (e.g., into another array or function call arguments).
>
> **Destructuring Assignment:** Destructuring allows you to unpack values from arrays or properties from objects into distinct variables with a concise syntax. For arrays, you use square brackets on the left side: `const [a, b] = [1, 2]` will assign `a = 1, b = 2`. For objects, use curly braces: `const {name, age} = person` will extract the `name` and `age` properties from the `person` object into same-named variables. You can destructure multiple values at once, provide default values, and even rename variables (e.g., `const {name: userName} = person`). Destructuring can be combined with rest: in an array, you might pick the first element and gather the rest (`const [head, ...tail] = list`), and in an object, gather remaining properties (`const {id, ...others} = data`).

Let's see a practical example that uses spread and destructuring for both arrays and objects:

```
// Array spread and destructuring
const numbers = [1, 2, 3];
const newNumbers = [...numbers, 4, 5];
console.log(newNumbers);            // -> [ 1, 2, 3, 4, 5 ]

const [first, ...restNums] = newNumbers;
console.log(first);                 // -> 1
console.log(restNums);              // -> [ 2, 3, 4, 5 ]

// Object spread and destructuring
const person = { name: 'Alice', age: 25, city: 'London' };
const clone = { ...person };  // clone the object using spread
console.log(clone);           // -> { name: 'Alice', age: 25, city: 'London' }

const { name, ...details } = person;
console.log(name);            // -> Alice
console.log(details);         // -> { age: 25, city: 'London' }
```

In the array portion above, we started with `numbers = [1,2,3]` and used `...numbers` inside a new array literal to spread its elements out, effectively creating `[1,2,3,4,5]`. We then destructured that array: `first` receives the first element `1`, and `restNums` gathers the remaining elements `[2,3,4,5]` due to the rest `...` in the destructuring pattern.

In the object portion, we used `{ ...person }` to make a shallow copy of `person`. This is often used to clone objects or merge multiple objects together. Then, using object destructuring, we pulled out the `name` property into a variable, and used `...details` to collect the remaining properties (age and city) into a `details` object. After this operation, `name` holds `"Alice"`, and `details` is `{ age: 25, city: 'London' }`. Note that the original `person` object is not modified by either the spread or destructuring.

These features greatly simplify tasks like splitting arrays, copying objects, or picking apart structures for convenience. They help make code more readable and reduce the need for manual loops or verbose `Object.assign` calls.

Exercise: Given an array of numbers, use the spread operator to add a new number to the end of the array without modifying the original array. Then use destructuring to extract the first two numbers into variables and the rest into a separate array. Similarly, for an object, try cloning the object with spread, then use destructuring to pick out one specific property (e.g., `id`) and gather all other properties into a new object.

3.4 Variadic functions and recursion (Fibonacci example)

In JavaScript (as in many languages), functions can be defined to handle a **variable number of arguments**, known as variadic functions. We touched on this with rest parameters (`...args`) which make it easy to capture arbitrarily many arguments into an array. Before rest parameters, JavaScript had an `arguments` object available inside any function, which was array-like and contained all passed arguments. However, `arguments` is not as convenient or safe as using a formal rest parameter. A variadic function can be useful for tasks like summing an arbitrary quantity of numbers, concatenating an arbitrary number of strings, etc.

Recursion is a fundamental programming concept where a function calls itself to solve a smaller sub-problem of a larger task. If you are experienced in other languages, you have likely seen recursive definitions (e.g., computing factorials or Fibonacci numbers). In JavaScript, recursion works the same way. The key is to define a **base case** (a condition where the function returns a value without calling itself) and a **recursive case** (where the function calls itself with a smaller or simpler input). JavaScript function calls use a stack, so deeply recursive algorithms may run into stack limits, but for many problems recursion provides a clear and elegant solution.

Below we demonstrate both variadic function use and recursion. First, we write a variadic sum function using rest parameters. Then, we implement a recursive function to compute Fibonacci numbers. (The Fibonacci sequence is a classic example where `F(n) = F(n-1) + F(n-2)` with base cases `F(0)=0, F(1)=1`.)

```
// Variadic function: sum all arguments
function sumAll(...numbers) {
    let total = 0;
    for (const n of numbers) {
        total += n;
    }
    return total;
}
console.log(sumAll(5, 10, 15));  // -> 30
console.log(sumAll(2, -1, 3, 7)); // -> 11
console.log(sumAll());           // -> 0  (no arguments returns 0)
```

```
// Recursive function: Fibonacci sequence
function fib(n) {
    if (n <= 1) {
        return n;               // base cases: fib(0)=0, fib(1)=1
    }
    return fib(n - 1) + fib(n - 2);
}
console.log(fib(6));  // -> 8
```

In the `sumAll` function, the syntax `...numbers` in the parameter list collects all passed arguments into the array `numbers`. We initialize a `total` and simply loop through the array, adding each number to `total`. This works for any number of arguments: `sumAll(5, 10, 15)` returns 30, and it even works when no arguments are passed (the loop runs zero times, leaving `total` as 0). This approach saves us from writing multiple function overloads or manually checking the `arguments` object.

For the `fib(n)` function, notice the structure: it calls itself for `n-1` and `n-2`. When `n` is 0 or 1, it immediately returns `n` (the base case). For larger `n`, it breaks the problem down by summing the results of smaller Fibonacci computations. For example, `fib(6)` triggers a chain of calls: `fib(6)` calls `fib(5)` and `fib(4)`; `fib(5)` calls `fib(4)` and `fib(3)`; and so on until reaching `fib(1)` and `fib(0)` which return immediately. The results then build back up: `fib(2)` returns 1, `fib(3)` returns 2, up to `fib(6)` returning 8. This example shows a clear recursive structure, albeit not the most efficient way to compute Fibonacci numbers for large `n` (due to repeated calculations). It's fine here for educational purposes and small inputs.

Exercise: Write your own variadic function, for example `multiplyAll(...nums)` that returns the product of an arbitrary quantity of numbers. Test it with different counts of arguments (including none). Additionally, try writing a recursive function for a different task, such as computing the factorial of a number `n` (recall that factorial *n!* = 1 × 2 × ... × n, with the base case 0! = 1). Ensure your recursive function has a clear base case and returns expected values for both small and larger inputs.

3.5 Sorting Arrays with `localeCompare`

Sorting arrays of strings in JavaScript can sometimes yield results that surprise developers coming from other languages. The default `Array.prototype.sort()` in JavaScript, when used on strings, sorts elements by their UTF-16 code unit values (essentially the character encoding values). This means uppercase letters, lowercase letters, and characters with accents might not end up in the order one might consider "alphabetical". For example, by default `"Z"` comes before `"a"` because the code point for `Z` (90) is less than that for `a` (97). Additionally, characters like `"É"` or `"ü"` may be sorted after plain ASCII letters since their code points are higher.

To perform a more natural, locale-aware sort, we can use the string method **`localeCompare()`** in the sorting callback. The `localeCompare` method compares two strings according to the language-specific rules (by default, the runtime's locale, or you can specify a locale). It returns a negative number if the reference string comes before the compared string, a positive number if after, or 0 if they are considered equivalent in sort order. By using `localeCompare` in the sort callback, we delegate string comparison to this method, which handles case and accent differences more intelligently.

Consider the following example:

```
const names = ["Zoe", "Émile", "john", "Alice"];
names.sort();
console.log("Default sort:", names);
// -> Default sort: [ 'Alice', 'Zoe', 'john', 'Émile' ]

names.sort((a, b) => a.localeCompare(b));
console.log("Sorted with localeCompare:", names);
// -> Sorted with localeCompare: [ 'Alice', 'Émile', 'john', 'Zoe' ]
```

In the default sort, you can see the order might not match typical alphabetic expectations: `"Alice"` (capital A) comes first, but `"Zoe"` (capital Z) comes before `"john"` (lowercase j) due to uppercase letters all sorting before any lowercase letters. `"Émile"` (capital E with accent) ends up at the end because its code point is beyond the basic Latin letters. After using `localeCompare`, the order becomes more intuitive: `"Alice"` (A) comes first, `"Émile"` (É, treated like E) comes next, followed by `"john"` (j) and `"Zoe"` (Z) at the end. Depending on locale settings, `localeCompare` might also handle accented characters as variants of their base letter (É vs E) and can sort case-insensitively by default.

Using `localeCompare` is especially important for internationalised applications, where data might include names or words with accents and you want sorting to respect the local alphabetical order. It's also useful to ensure a case-insensitive sort (by converting strings to a common case or using `localeCompare` with proper options).

Exercise: Create an array of mixed-case strings (e.g. `["banana", "Apple", "cherry", "apple"]`). First, sort it using the default `sort()` and observe the order. Then, sort it using `localeCompare` (e.g. `arr.sort((a, b) => a.localeCompare(b))`). Observe how the order changes. For a more advanced exercise, try providing localeCompare with a specific locale, for example sorting an array of French words with accents using `localeCompare('fr')`, and see if the order matches French alphabetical ordering.

3.6 Command-Line Arguments with `process.argv`

When running JavaScript with Node.js, we often need to accept input from the command line. In Node, any arguments provided after the script name in the command (`node script.js arg1 arg2 ...`) are made available via the **`process.argv`** array. If you have prior experience in languages like C (with `argv`), Python (`sys.argv`), or Java (the `args` array in `main`), the idea is similar.

In `process.argv`, the first element (`process.argv[0]`) is the path to the Node executable, the second element (`process.argv[1]`) is the path to the script file being executed, and the rest (`process.argv[2]` onward) are the actual command-line arguments the user provided. Typically, you will ignore the first two and focus on `process.argv.slice(2)` for the meaningful arguments.

Here is a simple example script that prints out the arguments it receives:

```
// args-demo.js
const args = process.argv.slice(2);
console.log("Arguments:", args);
```

If we run this script via the Node.js command line and provide some arguments, we can see how it works:

```
$ node args-demo.js apple banana cherry
Arguments: [ 'apple', 'banana', 'cherry' ]
```

In the example above, `process.argv.slice(2)` produces an array of `['apple', 'banana', 'cherry']` because those were the extra arguments after the script name. The script then logs this array. Each element in the array is a string. If you were expecting numeric inputs (for example, if the user is meant to provide numbers), remember that these will still come through as strings and you may need to convert them with `Number()` or parse them as integers.

For a more advanced usage, you might iterate over `process.argv.slice(2)` or use a library to parse flags (for complex CLI interfaces). But at its core, `process.argv` provides a straightforward way to get raw command-line inputs. Many simple Node scripts and tools use this to read filenames, configuration flags, or other data from the user when launching the script.

Exercise: Write a Node script (for example, `sumArgs.js`) that reads two numbers from the command line arguments (use `process.argv[2]` and `[3]`), converts them to numbers, adds them, and prints the

result. Test your script by running `node sumArgs.js 5 7` (which should output 12). Next, try extending it to handle an arbitrary number of numeric arguments (hint: use `slice(2)` to get all arguments and then use a loop or the `sumAll` function you wrote earlier to sum them).

3.7 Character Frequency Analysis with `Array.reduce()`

Functional programming methods like **`Array.prototype.reduce()`** allow you to perform powerful computations on arrays with minimal code. The `reduce()` method iterates over an array, applying a reducer function to each element to accumulate a single result. This is particularly handy for tasks like summing values, merging arrays of arrays, or aggregating object properties. Here, we will use `reduce()` to perform a character frequency analysis on a string — essentially counting how many times each character appears.

The idea is straightforward: take a string, split it into an array of characters, then reduce that array into an object where keys are characters and values are counts. Each iteration of `reduce` will either increment the count for a character if it's seen before or initialize it to 1 if it's the first occurrence. We will ignore spaces (or we could handle them separately) to focus on letters.

```
const phrase = "hello world";
const frequencies = phrase.split('').reduce((counts, char) => {
    if (char !== ' ') {                    // skip spaces
        counts[char] = (counts[char] || 0) + 1;
    }
    return counts;
}, {});
console.log(frequencies);  // -> { h: 1, e: 1, l: 3, o: 2, w: 1, r: 1, d: 1 }
```

Let's break down the code: 1. `phrase.split('')` turns the string `"hello world"` into an array of single characters: `["h","e","l","l","o"," ","w","o","r","l","d"]`. 2. We call `reduce` on this array. The reducer function takes an accumulator (here we call it `counts`) and the current element (`char`). We initialize the accumulator as an empty object `{}` (that's the second argument to `reduce` after the function). 3. Inside the reducer, we check `if (char !== ' ')` to ignore the space character. If the character is not a space, we update the `counts` object: `counts[char] = (counts[char] || 0) + 1;`. This uses the fact that `counts[char] || 0` will yield the current count if already set, or `0` if that character key doesn't exist yet. Then we add 1 to it. 4. We return the `counts` object from the reducer so that it becomes the accumulator for the next iteration. 5. After the reduce finishes, `frequencies` holds the final counts object. For `"hello world"`, the output object is `{ h: 1, e: 1, l: 3, o: 2, w: 1, r: 1, d: 1 }`. This indicates, for instance, that **l** appears 3 times, **o** appears 2 times, and all other letters in the phrase appear once.

Using `reduce` in this way can initially seem tricky, but it's very powerful. We defined *what* we want to do with each character (build up counts) without explicitly writing out a loop indexing over the array. This pattern can be adapted to many similar tasks, such as counting word frequencies in a paragraph (split by spaces and reduce), summing up values of object properties in an array of objects, and so on.

Exercise: Modify the above code to be case-insensitive, i.e., treat "A" and "a" as the same character when counting. (Hint: convert the string to all lower-case or upper-case before splitting.) Additionally, try using `reduce()` to count the occurrences of words in a sentence (you can split a string by spaces into words, then reduce to count each word). This will give you practice in transforming data and using `reduce` for aggregation.

What We Have Learned

- **Block vs Function Scope:** `let` and `const` create block-scoped variables, whereas `var` is function-scoped (or global-scoped if outside functions). Prefer `let/const` to avoid scope pitfalls and unintended re-declarations.
- **Arrow Functions:** Provide a concise syntax (`=>`) for writing functions. They capture the surrounding `this` context, cannot be used as constructors, and have no `arguments` object, distinguishing them from regular functions.
- **Rest and Spread (`...`):** The `...` syntax gathers multiple function arguments into an array (rest parameters) and spreads arrays/objects into lists of elements or key-value pairs. These simplify handling of variable arguments, array concatenation, and object copying.
- **Destructuring:** A convenient way to extract values from arrays or objects into variables. It allows pattern matching on data structures, often used alongside rest (to collect remaining items).
- **Variadic Functions:** Functions like `sumAll(...nums)` can accept any number of arguments, facilitated by rest parameters. This avoids manual handling of the `arguments` object.
- **Recursion:** Functions can call themselves to solve problems incrementally. We used a recursive definition for Fibonacci numbers as an example, illustrating base cases and recursive cases.
- **Sorting with localeCompare:** Using `string.localeCompare` in array sorting ensures alphabetical order that respects locale-specific rules and character case/accent ordering, in contrast to the default Unicode point sort.
- **Node.js CLI Arguments:** Node's `process.argv` array provides access to command-line arguments. By slicing off the first two elements, a script can easily read user-provided inputs (as strings) and act on them.
- **Array.reduce for Aggregation:** The `reduce()` method allows transformation of an array into a single accumulated result (number, array, object, etc.). We applied it to count character frequencies in a string, showcasing how to build an object incrementally via a reducer function.

Brief Recap

In this segment, we explored key features of modern JavaScript that form the foundation for effective coding. We learned how variable declarations have evolved with `let` and `const` bringing block scoping and immutability for constants. We practiced writing arrow functions for cleaner, more concise code and noted how their behaviour differs from traditional functions. We then delved into the versatile `...` syntax for rest parameters and spread operations, along with destructuring assignments to easily manipulate array and object data. Moving further, we discussed variadic functions and demonstrated recursion through a Fibonacci example, reinforcing how JavaScript handles function calls and returns. We also tackled practical tasks like sorting strings in a locale-aware manner using `localeCompare`, reading command-line arguments in Node.js via `process.argv`, and utilising the functional power of `reduce()` to perform aggregations (like counting character frequencies). Throughout these topics, we emphasized a hands-on understanding with code examples and encouraged experimentation with exercises.

With these core JavaScript concepts in hand, you are now equipped to write clearer and more robust code using modern syntax. You've also seen how Node.js allows your scripts to interact with the outside world (through command-line input), and how built-in methods enable complex data processing with relative ease.

As we move forward, we will build on this knowledge to explore how JavaScript interacts with web pages. So far, we've been working with JavaScript in isolation (or within Node.js). **Next, we will**

transition to JavaScript in the browser, focusing on the Document Object Model (DOM) and browser-specific APIs in Segment 4. This will open up the capability to manipulate web page content dynamically and handle user interactions, completing our journey from core language fundamentals to practical web development.