JavaScript Fundamentals Review

String Constructor and toString() Method

- **Definition**: A string object is used to represent a sequence of characters. String objects are created using the String constructor function, which wraps the primitive value in an object.

Example Code

```
const greetingObject = new String("Hello, world!");


console.log(typeof greetingObject); // "object"
```

- **toString() Method**: This method converts a value to its string representation. It is a method you can use for numbers, booleans, arrays, and objects.

Example Code

```
const num = 10;

console.log(num.toString()); // "10"


const arr = [1, 2, 3];

console.log(arr.toString()); // "1,2,3"
```

This method accepts an optional radix which is a number from 2 to 36. This radix represents the base, such as base 2 for binary or base 8 for octal. If the radix is not specified, it defaults to base 10, which is decimal.

Example Code

```
const num = 10;

console.log(num.toString(2)); // "1010"(binary)
```

Number Constructor

- **Definition**: The Number constructor is used to create a number object. The number object contains a few helpful properties and methods like

the isNaN and toFixed method. Most of the time, you will be using the Number constructor to convert other data types to the number data type.

Example Code

```
const myNum = new Number("34");

console.log(typeof myNum); // "object"


const num = Number('100');

console.log(num); // 100


console.log(typeof num); // number
```

Best Practices for Naming Variables and Functions

- **camelCasing**: By convention, JavaScript developers will use camel casing for naming variables and functions. Camel casing is where the first word is all lowercase and the following words start with a capital letter. Ex. isLoading.
- **Naming Booleans**: For boolean variables, it's a common practice to use prefixes such as "is", "has", or "can".

Example Code

```
let isLoading = true;

let hasPermission = false;

let canEdit = true;
```

- **Naming Functions**: For functions, the name should clearly indicate what the function does. For functions that return a boolean (often called predicates), you can use the same "is", "has", or "can" prefixes. When you have functions that retrieve data, it is common to start with the word "get". When you have functions that set data, it is common start with the word "set". For event handler functions, you might prefix with "handle" or suffix with "Handler".

Example Code

```
function getUserData() { /* ... */ }
```

```
function isValidEmail(email) { /* ... */ }


function getProductDetails(productId) { /* ... */ }


function setUserPreferences(preferences) { /* ... */ }


function handleClick() { /* ... */ }
```

- **Naming Variables Inside Loops**: When naming iterator variables in loops, it's common to use single letters like i, j, or k.

Example Code

```
for (let i = 0; i < array.length; i++) { /* ... */ }
```

Working with Sparse Arrays

- **Definition**: It is possible to have arrays with empty slots. Empty slots are defined as slots with nothing in them. This is different than array slots with the value of undefined. These types of arrays are known as sparse arrays.

Example Code

```
const sparseArray = [1, , , 4];

console.log(sparseArray.length); // 4
```

Linters and Formatters

- **Linters**: A linter is a static code analysis tool that flags programming errors, bugs, stylistic errors, and suspicious constructs. An example of a common linter would be ESLint.
- **Formatters**: Formatters are tools that automatically format your code to adhere to a specific style guide. An example of a common formatter would be Prettier.

Memory Management

- **Definition**: Memory management is the process of controlling the memory, allocating it when needed and freeing it up when it's no longer needed. JavaScript uses automatic memory management. This means that JavaScript (more specifically, the JavaScript engine in your web browser) takes care of memory

allocation and deallocation for you. You don't have to explicitly free up memory in your code. This automatic process is often called "garbage collection."

Closures

- **Definition**: A closure is a function that has access to variables in its outer (enclosing) lexical scope, even after the outer function has returned.

Example Code

```
function outerFunction(x) {
 let y = 10;
 function innerFunction() {
  console.log(x + y);
 }
 return innerFunction;
}


let closure = outerFunction(5);
closure(); // 15
```

var Keyword and Hoisting

- **Definition**: var was the original way to declare variables before 2015. But there were some issues that came with var in terms of scope, redeclaration and more. So that is why modern JavaScript programming uses let and const instead.
- **Redeclaring Variables with var**: If you try to redeclare a variable using let, then you would get a SyntaxError. But with var, you are allowed to redeclare a variable.

Example Code

```
// Uncaught SyntaxError: Identifier 'num' has already been declared
let num = 19;
let num = 18;
```

```
var myNum = 5;

var myNum = 10; // This is allowed and doesn't throw an error


console.log(myNum) // 10
```

- **var and Scope**: Variables declared with var inside a block (like an if statement or a for loop) are still accessible outside that block.

Example Code

```
if (true) {
  var num = 5;
}
console.log(num); // 5
```

- **Hoisting**: This is JavaScript's default behavior of moving declarations to the top of their respective scopes during the compilation phase before the code is executed. When you declare a variable using the var keyword, JavaScript hoists the declaration to the top of its scope.

Example Code

```
console.log(num); // undefined
var num = 5;
console.log(num); // 5
```

When you declare a function using the function declaration syntax, both the function name and the function body are hoisted. This means you can call a function before you've declared it in your code.

Example Code

```
sayHello(); // "Hello, World!"


function sayHello() {
  console.log("Hello, World!");
}
```

Variable declarations made with let or const are hoisted, but they are not initialized, and you can't access them before the actual declaration in your code. This behavior is often referred to as the "temporal dead zone".

Example Code

```
console.log(num); // Throws a ReferenceError

let num = 10;
```

Working with Imports, Exports and Modules

- **Module**: This is a self-contained unit of code that encapsulates related functions, classes, or variables. To create a module, you write your JavaScript code in a separate file.
- **Exports**: Any variables, functions, or classes you want to make available to other parts of your application need to be explicitly exported using the export keyword. There are two types of export: named export and default export.
- **Imports**: To use the exported items in another part of your application, you need to import them using the import keyword. The types can be named import, default import, and namespace import.

Example Code

```
// Within a file called math.js, we export the following functions:


// Named export
export function add(num1, num2) {
  return num1 + num2;
}


// Default export
export default function subtract(num1, num2) {
  return num1 - num2;
}
```

```
// Within another file, we can import the functions from math.js.

// Named import - This line imports the add function.
// The name of the function must exactly match the one exported from math.js.
import { add } from './math.js';

// Default import - This line imports the subtract function.
// The name of the function can be anything.
import subtractFunc from './math.js';

// Namespace import - This line imports everything from the file.
import * as Math from './math.js';

console.log(add(5, 3)); // 8
console.log(subtractFunc(5, 3)); // 2
console.log(Math.add(5, 3)); // 8
console.log(Math.subtract(5, 3)); // 2
```