Grammar, Variables & Scope

Basics

JavaScript borrows most of its syntax from Java, C, and C++, but it has also been influenced by Awk, Perl, and Python.

JavaScript is **case-sensitive** and uses the **Unicode** character set. For example, the word Früh (which means "early" in German) could be used as a variable name.

```
const Früh = "foobar";
```

But, the variable `früh` is not the same as `Früh` because JavaScript is case sensitive.

In JavaScript, instructions are called **statements** and are separated by semicolons (;).
A semicolon is not necessary after a statement if it is written on its own line. But if more than one statement on a line is desired, then they *must* be separated by semicolons.
It is considered best practice, however, to always write a semicolon after a statement, even when it is not strictly needed.

Comments

The syntax of **comments** is the same as in C++ and in many other languages:

```
// a one-line comment

/* this is a longer,
 * multi-line comment
 */
```

You can't nest block comments.
Comments behave like whitespace and are discarded during script execution.

You might also see a third type of comment syntax at the start of some JavaScript files, which looks something like this:

```
#!/usr/bin/env node.
```

This is called **hashbang comment** syntax and is a special comment used to specify the path to a particular JavaScript engine that should execute the script.

Declarations

JavaScript has three kinds of variable declarations:

1. `var` - declares a variable, optionally initializing it to a value.
2. `let` - declares a block-scoped, local variable, optionally initializing it to a value.
3. `const` - declares a block-scoped, read-only named constant.

Variables

You use variables as symbolic names for values in your application. The names of variables, called **identifiers**, conform to certain rules.

A JavaScript identifier usually starts with a letter, underscore (_), or a dollar sign ($). Subsequent characters can also be digits (0 – 9). Because JavaScript is case-sensitive, letters include the characters A through Z (uppercase) as well as a through z (lowercase).

Some examples of legal names are `Number_hits`, `temp99`, `$credit`, and `_name`.

Declaring variables

To declare (create) a variable, we need to **use the var**, **let**, or **const** keyword, followed by the name we want to provide to the variable.

The **var**, **let**, and **const** keywords **tell JavaScript to set aside a portion of memory so that we may store a specific piece of data to it later**.

The **name provided to the variable can be later used to access the location in memory assigned to the variable and retrieve the data which is stored in it**. To assign a value to the variable (initialize the variable with a value), use the assignment operator **=** to set the variable name equal to a piece of data (number, boolean, string, array, object, function, etc.)

```
/ Declare a variable named "x" using the var keyword
var x;

// Declare a variable named "y" using the let keyword
let y;

// Declare a variable named "z" using the const keyword
// Assign a value of 2 to the variable "z" using the assignment operator (=)
// Also called initializing "z" with a value of 2 (see section below on initialization)
const z = 2;
```

You can declare variables to unpack values using the **destructuring assignment** syntax. For example, `const { bar } = foo`. This will create a variable named `bar` and assign to it the value corresponding to the key of the same name from our object `foo`.

Variables should always be declared before they are used. JavaScript used to allow assigning to undeclared variables, which creates an undeclared global variable. This is an error in strict mode and should be avoided altogether.

Initialization

In a statement like `let x = 42`, the let x part is called a **declaration**, and the `= 42` part is called an **initializer**. The declaration allows the variable to be accessed later in code without throwing a ReferenceError, while the initializer **assigns a value to the variable**. In var and let declarations, the initializer is optional. If a variable is declared without an initializer, it is assigned the value undefined.

```
let x;
console.log(x); // logs "undefined"
```

In essence, `let x = 42` is equivalent to `let x; x = 42`.

const declarations always need an initializer, because they forbid any kind of assignment after declaration, and implicitly initializing it with `undefined` is likely a programmer mistake.

```
const x; // SyntaxError: Missing initializer in const declaration
```

Variable scope

In JavaScript, there are 4 types of variable scope, namely:

1. Global Scope - The default scope for all code running in script mode.
2. Module Scope - The scope for code running in module mode.
3. Function Scope - The scope created with a function.

In addition, variables declared with let or const can belong to an additional scope:

4. Block Scope - The scope created with a pair of curly braces (a block)

Global Scope

The **variables defined outside of any function or curly brackets** are known as global variables and have global scope. Global scope means that the variables can be accessed from any part of that program, any function or conditional state can access that variable.

For example:

```
const x = "declared outside function";

exampleFunction();

function exampleFunction() {
  console.log("Inside function");
  console.log(x);
}
```

```
console.log("Outside function");
console.log(x);
```

Note: It is not a good practice to use global variables when they are not needed as every code block will have access to those variables.

Local Scope

If you were to define some variables inside curly brackets {} or inside a specific function then those variables are called local variables. But, with the release of ES6, the local scope was further broken down into two different scopes:

- Function scope
- Block scope.

Function Scope

The function scope is the accessibility of the **variables defined inside a function**, these variables **cannot be accessed from any other function** or even outside the function in the main file.

For example:

```
function abc() {
year = 2021;
// the "year" variable can be accessed inside this function
console.log("The year is "+ Year);
}
// the "year" variable cannot be accessed outside here
abc();
```

Block Scope

Block scope is also a sub-type of local scope. The block scope can be defined as the scope of the variables inside the curly brackets {}. Now, these curly brackets can be of loops, or conditional statements, or something else. **You are only allowed to refer to these variables from within the curly brackets {}**. However, variables created with var are not block-scoped, but only local to the function (or global scope) that the block resides within.

Imagine a nested situation:

A block of code enclosed with curly brackets {} containing some block variables.

```
{
  let a = 10;
  const b = 5;
}
```

This block of code is itself inside a function.

```
function addition() {
  {
    let a = 10;
    const b = 5;
  }
}
```

Now when we try to access the variables from inside the function but outside of that specific block.

```
function addition() {
  {
    let a = 10;
    const b = 5;
  }
  console.log(a + b);
}
```

And to access this function we need to invoke it by:

```
addition();
```

Then **we'll be met with an error**, even though they are local variables of the same function.

The keywords **let** and **const** are used to define block variables.
If you are trying to refer to the local variable from outside the function you will get an error **"status (variable name) is not defined".**

Importance of let and const keywords while declaring block variables

If you declare a variable inside **curly brackets {}** without using the **let** and the **const** keywords then the variable will be declared as a "**Global variable**". This is due to the fact keywords have their scopes predefined.

Global variables

Global variables are in fact **properties of the global object**.

In web pages, the global object is `window`, so you can set and access global variables using the `window.variable syntax`. In all environments, you can use the `globalThis` variable (which itself is a global variable) to access global variables.

Consequently, you can access global variables declared in one window or frame from another window or frame by specifying the `window` or `frame` name. For example, if a variable called `phoneNumber` is declared in a document, you can refer to this variable from an `iframe` as `parent.phoneNumber`.

Variable hoisting

JavaScript only hoists declarations, not initializations!

This means that initialization doesn't happen until the associated line of code is executed. Until that point in the execution is reached the variable has its *default* initialization (undefined for a variable declared using var, otherwise uninitialized).

var hoisting

Here we declare and then initialize the value of a `var` after using it. **The default initialization of the `var` is `undefined`**.

```
console.log(num); // Returns 'undefined' from hoisted var declaration (not 6)
var num; // Declaration
num = 6; // Initialization
console.log(num); // Returns 6 after the line with initialization is executed.
```

The same thing happens if we declare and initialize the variable in the same line.

```
console.log(num); // Returns 'undefined' from hoisted var declaration (not 6)
var num = 6; // Initialization and declaration.
console.log(num); // Returns 6 after the line with initialization is executed.
```

If we forget the declaration altogether (and only initialize the value) the variable isn't hoisted. Trying to read the variable before it is initialized results in a ReferenceError exception.

```
console.log(num); /* Throws ReferenceError exception
                     - the interpreter doesn't know about `num` */
num = 6; // Initialization
```

Note however that initialization also causes declaration (if not already declared). The code snippet below will work, because even though it isn't hoisted, the variable is initialized and effectively declared before it is used.

```
a = "Cran"; // Initialize a
b = "berry"; // Initialize b

console.log(`${a}${b}`); // 'Cranberry'
```

let and const hoisting

Variables declared with let and const are also **hoisted but**, unlike var, are **not initialized with a default value**. An exception will be thrown if a variable declared with let or const is read before it is initialized.

```
console.log(num); /* Throws ReferenceError exception as the variable value
                     is uninitialized */
let num = 6; // Initialization
```

Note that it is the order in which code is executed that matters, not the order in which it is written in the source file. The code will succeed provided the line that initializes the variable is executed before any line that reads it.

Constants

You can create a read-only, named constant with the `const` keyword.
A constant cannot change value through assignment or be re-declared while the script is running.

However, **`const` only prevents *re-assignments* but doesn't prevent *mutations***. The properties of objects assigned to constants are not protected, so the following statement is executed without problems.

```
const MY_OBJECT = { key: 'value' };
MY_OBJECT.key = 'otherValue';
```

Also, the contents of an array are not protected, so the following statement is executed without problems.

```
const MY_ARRAY = ['HTML','CSS'];
MY_ARRAY.push('JAVASCRIPT');
console.log(MY_ARRAY); // ['HTML', 'CSS', 'JAVASCRIPT'];
```

Difference between var, let, and const keyword

1. Scope

var	let	const
Variables declared with `var` are in the **function scope**.	Variables declared as `let` are in the **block scope**.	Variables declared as `const` are in the **block scope**.

`function f1(){`
`  var a=10;`
`}`
`console.log(a)`
	`for (let i = 0; i < 3; i++){`
	`  console.log(i);`
	`}`
	`console.log(i);`
		`{`
		`  const x = 2;`
		`  console.log(x);`
		`}`
		`console.log(x);`

- In the var tab, when you will run the code, you will see an error as `a` is not defined because var variables are only accessible in the function scope.
- In the let tab, when you run the code, you will see an error as `i` is not defined because let variables are only accessible within the block they are declared.
- In the const tab, when you run the code, you will see that `x` was defined under the block was accessible, but when you try to access `x` outside that block, you will get an error.

2. Hoisting

Hoisting means that you can define a variable before its declaration.

var	let	const
Allowed	Not allowed	Not allowed

`x = 8;`
`console.log(x);`
`var x;`
	`x = 8;`
	`console.log(x);`
	`let x;`
		`x = 8;`
		`console.log(x);`
		`const x;`

- In the var tab, when you run the code, you will see there is no error and that we can declare var variables after defining them.
- In the let tab, when you run the code, you will get an error as let variables do not support hoisting.
- In the const tab, when you run the code, you will get an error as const variables do not support hoisting.

3. Reassign the value

To reassign a value is to reassign the value of a variable.

var	let	const
Allowed	Allowed	Not allowed

`var v1 = 1;`
`v1 = 30;`
`console.log(v1);`
	`let v1 = 1;`
	`v1 = 30;`
	`console.log(v1);`
		`const v1 = 1;`
		`v1 = 30;`
		`console.log(v1);`

- In the var and let tab, when you run the code, you will see that there is no error and we can define new values to the var and let variables.

- In the const tab, when you run the code, you will get an error as a const variables value cannot be reassigned.

4. Redeclaration of the variable

The redeclaration of a variable means that you can declare the variable again.

var	let	const
Allowed	Not allowed	Not allowed

`var v1 = 1;`
`var v1 = 30;`
`console.log(v1);`
	`let v1 = 1;`
	`let v1 = 30;`
	`console.log(v1);`
		`const v1 = 1;`
		`const v1 = 30;`
		`console.log(v1);`

- In the var tab, when you run the code, you will see that there is no error as we allowed to declare the same variable again.
- In the let and the const tabs, when you run the code, you will get an error as the let and const variables do not allow you to redeclare them again.