**Assignment Day 1**

Question 1

Ans:

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | |
| [assert()](https://www.w3schools.com/jsref/met_console_assert.asp) | Writes an error message to the console if the assertion is false | |
| [clear()](https://www.w3schools.com/jsref/met_console_clear.asp) | Clears the console | |
| [count()](https://www.w3schools.com/jsref/met_console_count.asp) | Logs the number of times that this particular call to count() has been called | |
| [error()](https://www.w3schools.com/jsref/met_console_error.asp) | | Outputs an error message to the console |
| [group()](https://www.w3schools.com/jsref/met_console_group.asp) | | Creates a new inline group in the console. This indents following console messages by an additional level, until console.groupEnd() is called |
| [groupCollapsed()](https://www.w3schools.com/jsref/met_console_groupcollapsed.asp) | | Creates a new inline group in the console. However, the new group is created collapsed. The user will need to use the disclosure button to expand it |
| [groupEnd()](https://www.w3schools.com/jsref/met_console_groupend.asp) | | Exits the current inline group in the console |
| [info()](https://www.w3schools.com/jsref/met_console_info.asp) | | Outputs an informational message to the console |
| [log()](https://www.w3schools.com/jsref/met_console_log.asp) | | Outputs a message to the console |
| [table()](https://www.w3schools.com/jsref/met_console_table.asp) | | Displays tabular data as a table |
| [time()](https://www.w3schools.com/jsref/met_console_time.asp) | | Starts a timer (can track how long an operation takes) |
| [timeEnd()](https://www.w3schools.com/jsref/met_console_timeend.asp) | | Stops a timer that was previously started by console.time() |
| [trace()](https://www.w3schools.com/jsref/met_console_trace.asp) | | Outputs a stack trace to the console |
| [warn()](https://www.w3schools.com/jsref/met_console_warn.asp) | | Outputs a warning message to the console |

Question 2

Ans:

**Var**

Before the advent of ES6, var declarations ruled. There are issues associated with variables declared with var, though. That is why it was necessary for new ways to declare variables to emerge.

**Scope of var**

**Scope** essentially means where these variables are available for use. var declarations are globally scoped or function/locally scoped.

The scope is global when a var variable is declared outside a function. This means that any variable that is declared with var outside a function block is available for use in the whole window.

var is function scoped when it is declared within a function. This means that it is available and can be accessed only within that function.

To understand further, look at the example below.

var greeter = "hey hi";

function newFunction() {

var hello = "hello";

}

Here, greeter is globally scoped because it exists outside a function while hello is function scoped. So we cannot access the variable hello outside of a function. So if we do this:

var tester = "hey hi";

function newFunction() {

var hello = "hello";

}

console.log(hello); // error: hello is not defined

We'll get an error which is as a result of hello not being available outside the function.

### var variables can be re-declared and updated

This means that we can do this within the same scope and won't get an error.

var greeter = "hey hi";

var greeter = "say Hello instead";

and this also

var greeter = "hey hi";

greeter = "say Hello instead";

### Hoisting of var

Hoisting is a JavaScript mechanism where variables and function declarations are moved to the top of their scope before code execution. This means that if we do this:

console.log (greeter);

var greeter = "say hello"

it is interpreted as this:

var greeter;

console.log(greeter); // greeter is undefined

greeter = "say hello"

So var variables are hoisted to the top of their scope and initialized with a value of undefined.

### Problem with var

There's a weakness that comes with  var. I'll use the example below to explain:

var greeter = "hey hi";

var times = 4;

if (times > 3) {

var greeter = "say Hello instead";

}

console.log(greeter) // "say Hello instead"

So, since times > 3 returns true, greeter is redefined  to "say Hello instead". While this is not a problem if you knowingly want greeter to be redefined, it becomes a problem when you do not realize that a variable greeter has already been defined before.

If you have used greeter in other parts of your code, you might be surprised at the output you might get. This will likely cause a lot of bugs in your code. This is why let and const are necessary.

## Let

let is now preferred for variable declaration. It's no surprise as it comes as an improvement to var declarations. It also solves the problem with var that we just covered.

### let is block scoped

A block is a chunk of code bounded by {}. A block lives in curly braces. Anything within curly braces is a block.

So a variable declared in a block with let  is only available for use within that block.

let greeting = "say Hi";

let times = 4;

if (times > 3) {

let hello = "say Hello instead";

console.log(hello);// "say Hello instead"

}

console.log(hello) // hello is not defined

We see that using hello outside its block (the curly braces where it was defined) returns an error. This is because let variables are block scoped .

### let can be updated but not re-declared.

Just like var,  a variable declared with let can be updated within its scope. Unlike var, a let variable cannot be re-declared within its scope. So while this will work:

let greeting = "say Hi";

greeting = "say Hello instead";

this will return an error:

let greeting = "say Hi";

let greeting = "say Hello instead"; // error: Identifier 'greeting' has already been declared

However, if the same variable is defined in different scopes, there will be no error:

let greeting = "say Hi";

if (true) {

let greeting = "say Hello instead";

console.log(greeting); // "say Hello instead"

}

console.log(greeting); // "say Hi"

Why is there no error? This is because both instances are treated as different variables since they have different scopes.

This fact makes let a better choice than var. When using let, you don't have to bother if you have used a name for a variable before as a variable exists only within its scope.

Also, since a variable cannot be declared more than once within a scope, then the problem discussed earlier that occurs with var does not happen.

### Hoisting of let

Just like  var, let declarations are hoisted to the top. Unlike var which is initialized as undefined, the let keyword is not initialized. So if you try to use a let variable before declaration, you'll get a Reference Error.

## Const

Variables declared with the const maintain constant values. const declarations share some similarities with let declarations.

### const declarations are block scoped

Like let declarations, const declarations can only be accessed within the block they were declared.

### const cannot be updated or re-declared

This means that the value of a variable declared with const remains the same within its scope. It cannot be updated or re-declared. So if we declare a variable with const, we can neither do this:

const greeting = "say Hi";

greeting = "say Hello instead";// error: Assignment to constant variable.

nor this:

const greeting = "say Hi";

const greeting = "say Hello instead";// error: Identifier 'greeting' has already been declared

Every const declaration, therefore, must be initialized at the time of declaration.

This behavior is somehow different when it comes to objects declared with const. While a const object cannot be updated, the properties of this objects can be updated. Therefore, if we declare a const object as this:

const greeting = {

message: "say Hi",

times: 4

}

while we cannot do this:

const greeting = {

words: "Hello",

number: "five"

} // error: Assignment to constant variable.

we can do this:

greeting.message = "say Hello instead";

This will update the value of greeting.message without returning errors.

### Hoisting of const

Just like let, const declarations are hoisted to the top but are not initialized.

Major differences:

* var declarations are globally scoped or function scoped while let and const are block scoped.
* var variables can be updated and re-declared within its scope; let variables can be updated but not re-declared; const variables can neither be updated nor re-declared.
* They are all hoisted to the top of their scope. But while var variables are initialized with undefined, let and const variables are not initialized.
* While var and let can be declared without being initialized, const must be initialized during declaration.

Question 3:

Ans:

# Data types

A value in JavaScript is always of a certain type. For example, a string or a number.

There are eight basic data types in JavaScript. Here, we’ll cover them in general and in the next chapters we’ll talk about each of them in detail.

We can put any type in a variable. For example, a variable can at one moment be a string and then store a number:

// no error

let message = "hello";

message = 123456;

Programming languages that allow such things, such as JavaScript, are called “dynamically typed”, meaning that there exist data types, but variables are not bound to any of them.

## [Number](https://javascript.info/types" \l "number)

let n = 123;

n = 12.345;

The number type represents both integer and floating point numbers.

There are many operations for numbers, e.g. multiplication \*, division /, addition +, subtraction -, and so on.

Besides regular numbers, there are so-called “special numeric values” which also belong to this data type: Infinity, -Infinity and NaN.

* Infinity represents the mathematical [Infinity](https://en.wikipedia.org/wiki/Infinity) ∞. It is a special value that’s greater than any number.

We can get it as a result of division by zero:

alert( 1 / 0 ); // Infinity

Or just reference it directly:

alert( Infinity ); // Infinity

* NaN represents a computational error. It is a result of an incorrect or an undefined mathematical operation, for instance:

alert( "not a number" / 2 ); // NaN, such division is erroneous

NaN is sticky. Any further operation on NaN returns NaN:

alert( "not a number" / 2 + 5 ); // NaN

So, if there’s a NaN somewhere in a mathematical expression, it propagates to the whole result.

**Mathematical operations are safe**

Doing maths is “safe” in JavaScript. We can do anything: divide by zero, treat non-numeric strings as numbers, etc.

The script will never stop with a fatal error (“die”). At worst, we’ll get NaN as the result.

Special numeric values formally belong to the “number” type. Of course they are not numbers in the common sense of this word.

We’ll see more about working with numbers in the chapter [Numbers](https://javascript.info/number).

## [BigInt](https://javascript.info/types" \l "bigint)

In JavaScript, the “number” type cannot represent integer values larger than (253-1) (that’s 9007199254740991), or less than -(-253-1) for negatives. It’s a technical limitation caused by their internal representation.

For most purposes that’s quite enough, but sometimes we need really big numbers, e.g. for cryptography or microsecond-precision timestamps.

BigInt type was recently added to the language to represent integers of arbitrary length.

A BigInt value is created by appending n to the end of an integer:

// the "n" at the end means it's a BigInt

const bigInt = 1234567890123456789012345678901234567890n;

As BigInt numbers are rarely needed, we don’t cover them here, but devoted them a separate chapter [BigInt](https://javascript.info/bigint). Read it when you need such big numbers.

**Compatability issues**

Right now BigInt is supported in Firefox/Chrome/Edge, but not in Safari/IE.

## [String](https://javascript.info/types" \l "string)

A string in JavaScript must be surrounded by quotes.

let str = "Hello";

let str2 = 'Single quotes are ok too';

let phrase = `can embed another ${str}`;

In JavaScript, there are 3 types of quotes.

1. Double quotes: "Hello".
2. Single quotes: 'Hello'.
3. Backticks: `Hello`.

Double and single quotes are “simple” quotes. There’s practically no difference between them in JavaScript.

Backticks are “extended functionality” quotes. They allow us to embed variables and expressions into a string by wrapping them in ${…}, for example:

let name = "John";

// embed a variable

alert( `Hello, ${name}!` ); // Hello, John!

// embed an expression

alert( `the result is ${1 + 2}` ); // the result is 3

The expression inside ${…} is evaluated and the result becomes a part of the string. We can put anything in there: a variable like name or an arithmetical expression like 1 + 2 or something more complex.

Please note that this can only be done in backticks. Other quotes don’t have this embedding functionality!

alert( "the result is ${1 + 2}" ); // the result is ${1 + 2} (double quotes do nothing)

We’ll cover strings more thoroughly in the chapter [Strings](https://javascript.info/string).

**There is no character type.**

In some languages, there is a special “character” type for a single character. For example, in the C language and in Java it is called “char”.

In JavaScript, there is no such type. There’s only one type: string. A string may consist of only one character or many of them.

## [Boolean (logical type)](https://javascript.info/types" \l "boolean-logical-type)

The boolean type has only two values: true and false.

This type is commonly used to store yes/no values: true means “yes, correct”, and false means “no, incorrect”.

For instance:

let nameFieldChecked = true; // yes, name field is checked

let ageFieldChecked = false; // no, age field is not checked

Boolean values also come as a result of comparisons:

let isGreater = 4 > 1;

alert( isGreater ); // true (the comparison result is "yes")

We’ll cover booleans more deeply in the chapter [Logical operators](https://javascript.info/logical-operators).

## [The “null” value](https://javascript.info/types" \l "the-null-value)

The special null value does not belong to any of the types described above.

It forms a separate type of its own which contains only the null value:

let age = null;

In JavaScript, null is not a “reference to a non-existing object” or a “null pointer” like in some other languages.

It’s just a special value which represents “nothing”, “empty” or “value unknown”.

The code above states that age is unknown.

## [The “undefined” value](https://javascript.info/types" \l "the-undefined-value)

The special value undefined also stands apart. It makes a type of its own, just like null.

The meaning of undefined is “value is not assigned”.

If a variable is declared, but not assigned, then its value is undefined:

let age;

alert(age); // shows "undefined"

Technically, it is possible to explicitly assign undefined to a variable:

let age = 100;

// change the value to undefined

age = undefined;

alert(age); // "undefined"

…But we don’t recommend doing that. Normally, one uses null to assign an “empty” or “unknown” value to a variable, while undefined is reserved as a default initial value for unassigned things.

## [Objects and Symbols](https://javascript.info/types" \l "objects-and-symbols)

The object type is special.

All other types are called “primitive” because their values can contain only a single thing (be it a string or a number or whatever). In contrast, objects are used to store collections of data and more complex entities.

Being that important, objects deserve a special treatment. We’ll deal with them later in the chapter [Objects](https://javascript.info/object), after we learn more about primitives.

The symbol type is used to create unique identifiers for objects. We have to mention it here for the sake of completeness, but also postpone the details till we know objects.

## [The typeof operator](https://javascript.info/types" \l "type-typeof)

The typeof operator returns the type of the argument. It’s useful when we want to process values of different types differently or just want to do a quick check.

It supports two forms of syntax:

1. As an operator: typeof x.
2. As a function: typeof(x).

In other words, it works with parentheses or without them. The result is the same.

typeof undefined // "undefined"

typeof 0 // "number"

typeof 10n // "bigint"

typeof true // "boolean"

typeof "foo" // "string"

typeof Symbol("id") // "symbol"

typeof Math // "object" (1)

typeof null // "object" (2)

typeof alert // "function" (3)