JavaScript

WEB

3 - JavaScript

Document summary

This text provides a comprehensive introduction to JavaScript, covering its educational objectives, differences between client and server environments, data types, operators, variables, functions, and regular expressions. It emphasizes the ability to write and understand JavaScript code in various contexts, including client-side and server-side. Concepts are explained concisely with practical examples, making it a useful resource for beginners and those looking to deepen their understanding of JavaScript.

Table of content

1. Educational objectives

- Cite all the environments where JavaScript can be executed.
- Can write JavaScript code in all the environments where it can be executed.
- Explain the difference between interpreted and compiled languages.
- Read and write basic JavaScript code.
- Read regular expressions.
- Explain the difference between var, let, and const.
- Explain the difference between == and ===.
- Explain the difference between null and undefined.

2. Introduction

JavaScript is a lightweight, interpreted, or just-in-time compiled programming language with first-class functions.

Interpreted The interpreter reads the source code and executes it directly. It does not require the compilation of the program into machine-code.

Just-in-time (JIT) compiled The interpreter compiles the hot parts of the source code into machine-code and executes it directly. The rest of the program is interpreted.

First-class functions Functions are treated like any other value. They can be stored in variables, passed as arguments to other functions, created within functions, and returned from functions.

2.1. Client and server-side JavaScript

2.1.1. Client-side JavaScript

Client-side JavaScript is executed in the user's browser. It is used to create dynamic web pages. It can be used to validate user input, create animations, and communicate with the server without reloading the page.

2.1.1.1. Adding JavaScript directly to HTML

```
<script type='text/javascript'>
  console.log('Hello, World!');
  document.writeln('Hello, World!')
</script>
```

2.1.1.2. Adding JavaScript to an external file

```
<script src='script.js'></script>
```

- The defer attribute is used to defer the execution of the script until the page has been loaded.
- The async attribute is used to load the script asynchronously.

2.1.2. Server-side JavaScript

Server-side JavaScript is executed on the server. It is used to create web applications, APIs, and server-side scripts. It can be used to interact with databases, send emails, and perform other server-side tasks. You can run server-side JavaScript using Node.js.

3. JavaScript types

3.1. Primitive types

- Undefined: Unique primitive value undefined
- Number: Real or integer number (e.g. 3.14, 42)
- Boolean: true or false
- String: Character sequence, whose literals begin and end with single or double - quotes (e.g. “HEIG-VD”, ‘hello’)
- BigInt: Arbitrary-precision integers, whose literals end with an n (e.g. - 9007199254740992n)
- Symbol: Globally unique values usable as identifiers or keys in objects (e.g. Symbol(), Symbol(“description”))
- Null: Unique value null

In a dynamic language you don’t specify the type when you declare a variable and the type of a variable can change.

3.2. Objects

- Object: Collection of key-value pairs

The symtax for creating an object is:

```
let person = {
  firstName: 'John',
  lastName: 'Doe',
  age: 30
};
```

Properties can be accessed using dot notation or bracket notation:

```
console.log(person.firstName);    // John
console.log(person['lastName']);  // Doe
```

3.3. Arrays

- Array: Ordered collection of values

The syntax for creating an array is:

```
let fruits = ['Apple', 'Banana', 'Cherry'];
```

Elements can be accessed using bracket notation:

```
console.log(fruits[0]);       // Apple
console.log(fruits.length);   // Banana
```

3.3.1. Methods on arrays

```
fruits.push("mango", "papaya");   // Appends new items
fruits.pop();                     // Removes and returns the last item
fruits.reverse();                 // Reverses the items' order
fruits.splice(2, 1, 'Orange');    // Replaces 1 elemnt at position 2 with 'Orange'
fruits.splice(1, 0, 'Peach');     // Inserts 'Peach' at index 1
```

4. Operators

4.1. Typeof operator

The typeof operator returns the type of a variable or expression.

```
console.log(typeof 42);          // number
console.log(typeof 'hello');     // string
console.log(typeof null);        // object
console.log(typeof [1, 2, 3]);   // object
```

4.2. Arithmetic operators

```
1 + 1;  // addition
1 - 1;  // subtraction
1 / 1;  // division
1 * 1;  // multiplication
1 % 1;  // modulo
1 ** 1; // exponentiation
```

4.3. String operators

```
"con" + "cat" + "e" + "nate";
`PI = ${Math.PI}`; // template literals (instead of: "PI = " + Math.PI)
```

In practice we should opt for template literals over string concatenation.

4.4. Assignment operators

```
let a = 1;

// arithmetic assignments
a += 1; // addition
a -= 1; // subtraction
a *= 1; // multiplication
a /= 1; // division
a %= 1; // modulo
a **= 1; // exponentiation
```

4.4.1. Unary operators

```
let a = 1;

// unary operators
a++; // increment
++a; // increment
a--; // decrement
--a; // decrement
```

4.5. Destructuring assignment

```
var [a, b] = [1, 2];
console.log(a); // 1
console.log(b); // 2

var [a, b, ...rest] = [1, 2, 3, 4, 5];
console.log(a); // 1
console.log(b); // 2
console.log(rest); // [3, 4, 5]

var {a, b} = {a: 1, b: 2};
console.log(a); // 1
console.log(b); // 2

var {a, b, ...rest} = {a: 1, b: 2, c: 3, d: 4};
console.log(a); // 1
```

```
console.log(b); // 2
console.log(rest); // {c: 3, d: 4}
```

4.6. Logical operators

```
!true          // false
true && false  // false
true || false  // true
true ? a : b   // a
```

4.7. Optional chaining

```
const adventurer = {
    name: 'Alice',
    cat: {
        name: 'Dinah'
    }
};
console.log(adventurer.dog?.name); // expected output: undefined
console.log(adventurer.cat?.name); // expected output: Dinah
```

In this example, if we had omitted the ? symbol, it would have failed with a TypeError: adventurer.dog is undefined.

4.8. Comparison operators

```
1 == 1;
1 != 2;
1 < 2;
2 > 1;
1 <= 1;
1 >= 1;
```

4.8.1. Automatic type conversion

Automatic type conversion is performed when comparing values of different types. It is at the root of many issues when using comparison operators.

```
"1" == 1 // true (!!!)
false == 0 // true
8 * null // 0
```

4.8.2. Strict equality

Strict equality compares two values for equality without type conversion.

```
"1" === 1 // false
"1" !== 1 // true
```

5. Variables

5.1. var

The var statement declares a **non-block-scoped** variable, optionally initializing it to a value. Its scope is its current execution context, i.e. either the enclosing function or, if outside any function, global. It can be re-declared.

```
var x = 1;
if (true) { var x = 2; }  // same variable
console.log(x);           // 2
```

5.2. let

The let statement declares a **block-scoped** local variable, optionally initializing it to a value. It can be re-assigned but not re-declared.

```
let x = 1;
{ let x = 2; }  // different variable (in a new scope)
console.log(x); // 1
let x = 1000;   // Error: redeclaration of let x
```

5.3. const

The const statement declares a **block-scoped** read-only named constant. It can be re-assigned but not re-declared.

```
const x = 1;
x = 2; // TypeError: Assignment to constant variable.
```

6. Functions

6.1. Conditional execution

```
let num = prompt("Enter a number");
if (num > 0) {
    alert(`${num} is positive`);
} else if (num < 0) {
    alert(`${num} is negative`);
} else {
    alert(`${num} is zero`);
}
```

6.2. Switch statement

```
let val = prompt("Enter a letter");
switch(val) {
  case "a":
    alert("a");
    break;
  case "b":
    alert("b");
    break;
  default:
    alert("Not a or b");
    break;
}
```

6.3. While and do-while loops

```
let num = 0;
while (num < 10) {
  console.log(num);
  num += 1;
}
let echo = "";
do {
    echo = prompt("Echo");
    console.log(echo);
} while (echo != "stop");
```

6.4. For loop

```
for (let num = 0; num < 10; num++) {
    console.log(num);
}
```

The for...in statement iterates over the enumerable properties of an object.

```
let obj = {a: 1, b: 2, c: 3};
for (let prop in obj) {
    console.log(prop, obj[prop]);
}
```

The for...of statement creates a loop iterating over iterable objects.

```
let nums = [0, 1, 2, 3, 4, 5, 6, 7, 8 , 9];
for (let num of nums) {
    console.log(num);
}
```

6.5. Break and continue

The **break** statement terminates the current loop.

The **continue** statement terminates the execution of the current iteration and continues the execution of the loop with the next iteration.

break and **continue** can also be used with labelled statements, but please don't.

6.6. Exception handling

```
try {
  variable; // ReferenceError: variable is not defined
} catch (error) {
  // Fails silently
}
```

Exceptions can be triggered using **throw** and **Error**:

```
throw new Error("AAHHARG!!!");
```

7. Function

7.1. Declaration notation

```
function square(x) {
  return x * x;
}
// or
var square = function(x) {
    return x * x;
}
```

7.2. Arrow functions

```
var square = x => x * x
// or
var square = (x) => {
  return x * x;
}
```

7.3. Recursion

```
function factorial(n) {
  return n == 1 ? n : n * factorial(n-1);
}
console.log(factorial(5)) // 5 * 4 * 3 * 2 * 1 = 120
```

.. as long as it does not overflow the call stack.

7.4. Higher-order functions

```
function greaterThan(n) {
    return m => m > n;
}
let greaterThan10 = greaterThan(10);
console.log(greaterThan10(11)); // true
```

8. Regular expressions

Regular expressions are patterns used to match character combinations in strings. They are created using the RegExp constructor or a literal notation.

```
const re1 = /ab+c/;
const re2 = new RegExp(/ab+c/);
```

A regular expression pattern must be surrounded by slashes (/). The pattern can include flags that specify how the search is performed.

- `Character Classes (., \s, \d, …)` that distinguish types of chararters (resp. any, whitespace or digit)
- `Character sets ([A-Z], [a-z], [0-9], [abc], …)` that match any of the enclosed - characters (resp. uppercase letters, lowercase letters, digits, and any of a, b or c)
- Either operator (x|y) that match either the left or right handside values
- `Quantifiers (\*, +, ?, {n}, {n,m})` that indicate the number of times an expression matches
- `Boundaries (^, \$)` that indicate the beginnings and endings of lines and words
- `Groups ((), (?<name>), (?:))` that extracts and remember (or not) information from the input
- `Assertions (x(?=y))` that helps at defining conditional expressions

8.1. Flags

```
const re1 = /ab+c/;   // no flag
const re2 = /ab+c/g;  // global search
const re3 = /ab+c/i;  // case-insensitive search
const re4 = /ab+c/m;  // multi-line search
const re5 = /ab+c/gi  // global case-insensitive search
```