

JavaScript

Logical Operators

There are four logical operators in JavaScript: `||` (OR), `&&` (AND), `!` (NOT), `??` (Nullish Coalescing). Here we cover the first three, the `??` operator is in the next article.

Although they are called “logical”, they can be applied to values of any type, not only boolean. Their result can also be of any type.

Let’s see the details.

`||` (OR)

The “OR” operator is represented with two vertical line symbols:

```
result = a || b;
```

Logical Operators

In classical programming, the logical OR is meant to manipulate boolean values only. If any of its arguments are true, it returns true, otherwise it returns false.

In JavaScript, the operator is a little bit trickier and more powerful. But first, let's see what happens with boolean values.

There are four possible logical combinations:

```
alert( true || true ); // true
```

```
alert( false || true ); // true
```

```
alert( true || false ); // true
```

```
alert( false || false ); // false
```

Logical Operators

Most of the time, OR `||` is used in an if statement to test if any of the given conditions is true.

For example:

```
let hour = 9;
```

```
if (hour < 10 || hour > 18)
```

```
{alert( 'The office is closed.' );}
```

We can pass more conditions:

```
let hour = 12;
```

```
let isWeekend = true;
```

```
if (hour < 10 || hour > 18 || isWeekend)
```

```
{alert( 'The office is closed.' ); // it is the weekend}
```

Logical Operators

&& (AND)

The AND operator is represented with two ampersands &&:

```
result = a && b;
```

In classical programming, AND returns true if both operands are truthy and false otherwise:

```
alert( true && true ); // true
```

```
alert( false && true ); // false
```

```
alert( true && false ); // false
```

```
alert( false && false ); // false
```

Logical Operators

&& (AND)

An example with if:

```
let hour = 12;
```

```
let minute = 30;
```

```
if (hour == 12 && minute == 30)
```

```
{alert( 'The time is 12:30' );}
```

Logical Operators

! (NOT)

The boolean NOT operator is represented with an exclamation sign !.

The syntax is pretty simple:

```
result = !value;
```

The operator accepts a single argument and does the following:

1. Converts the operand to boolean type: true/false.
2. Returns the inverse value.

For instance:

```
alert( !true ); // false
```

```
alert( !0 ); // true
```

```
alert( !null ); // false
```

Task

Check the login

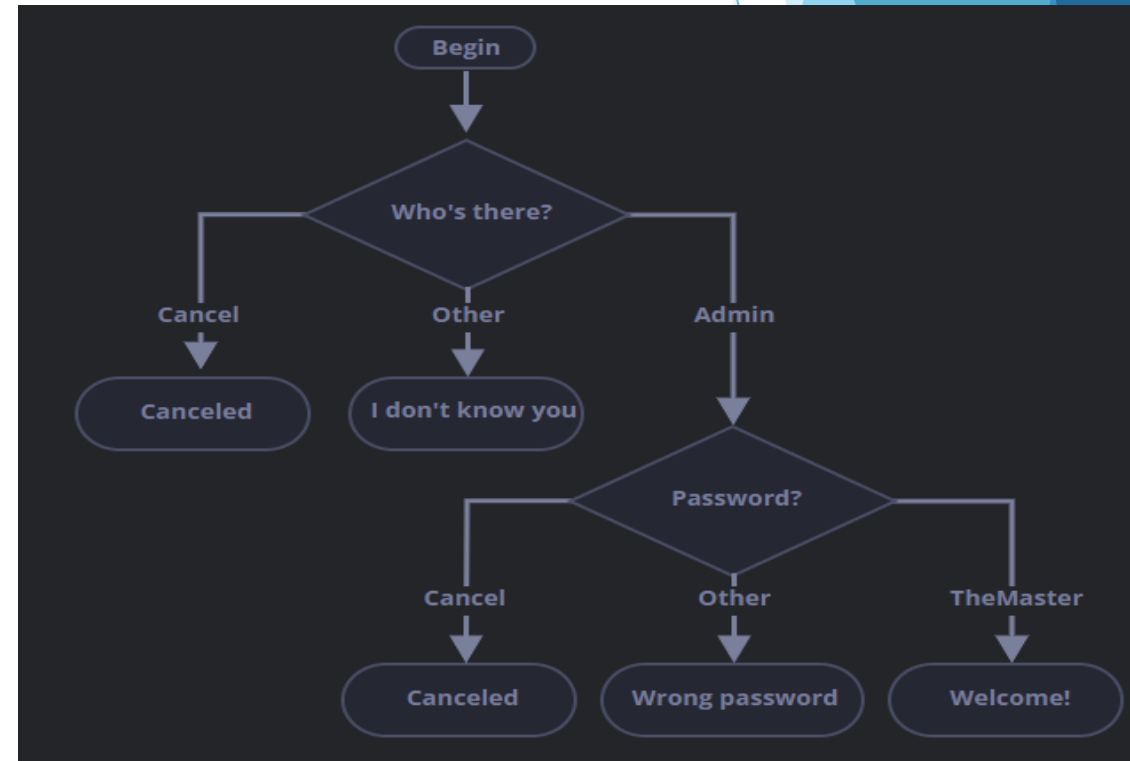
Write the code which asks for a login with prompt.

If the visitor enters "Admin", then prompt for a password, if the input is an empty line or Esc – show “Canceled”, if it’s another string – then show “I don’t know you”.

The password is checked as follows:

- If it equals “TheMaster”, then show “Welcome!”,
- Another string – show “Wrong password”,
- For an empty string or cancelled input, show

“Canceled”



Task solution

Check the login

```
let userName = prompt("Who's there?", '');

if (userName === 'Admin') {

    let pass = prompt('Password?', '');

    if (pass === 'TheMaster') {
        alert( 'Welcome!' );
    } else if (pass === '' || pass === null) {
        alert( 'Canceled' );
    } else {
        alert( 'Wrong password' );
    }

} else if (userName === '' || userName === null) {
    alert( 'Canceled' );
} else {
    alert( "I don't know you" );
}
```

Loops

Loops are a way to repeat the same code multiple times.

The “while” loop

The while loop has the following syntax:

```
while (condition) { // code // so-called "loop body" }
```

While the condition is truthy, the code from the loop body is executed.

For instance, the loop below outputs *i* while $i < 3$:

```
let i = 0;
```

```
while (i < 3) { // shows 0, then 1, then 2
```

```
  alert( i );
```

```
  i++; }
```

Loops

The “do...while” loop

The condition check can be moved below the loop body using the do..while syntax:

```
do { // loop body } while (condition);
```

The loop will first execute the body, then check the condition, and, while it's truthy, execute it again and again.

For example:

```
let i = 0;  
do {  
  alert( i );  
  i++; } while ( i < 3 );
```

Loops

The “for” loop

The for loop is more complex, but it’s also the most commonly used loop.

It looks like this:

```
for (begin; condition; step) { // ... loop body ... }
```

Let’s learn the meaning of these parts by example. The loop below runs `alert(i)` for `i` from 0 up to (but not including) 3:

```
for (let i = 0; i < 3; i++)
```

```
{ // shows 0, then 1, then 2
```

```
  alert(i);}
```

Loops

The “for” loop

```
for (let i = 0; i < 3; i++)  
{ // shows 0, then 1, then 2  
  alert(i);}
```

Let's examine the for statement part-by-part:

part		
begin	<code>let i = 0</code>	Executes once upon entering the loop.
condition	<code>i < 3</code>	Checked before every loop iteration. If false, the loop stops.
body	<code>alert(i)</code>	Runs again and again while the condition is truthy.
step	<code>i++</code>	Executes after the body on each iteration.

Loops

Breaking the loop

Normally, a loop exits when its condition becomes falsy.

But we can force the exit at any time using the special break directive.

For example, the loop below asks the user for a series of numbers, “breaking” when no number is entered:

```
let sum = 0;

while (true) {

  let value = +prompt("Enter a number", "");

  if (!value) break;

  sum += value;}

alert( 'Sum: ' + sum );
```

Loops

Continue to the next iteration

The continue directive is a “lighter version” of break. It doesn’t stop the whole loop. Instead, it stops the current iteration and forces the loop to start a new one (if the condition allows).

We can use it if we’re done with the current iteration and would like to move on to the next one.

The loop below uses continue to output only odd values:

```
for (let i = 0; i < 10; i++) {  
  // if true, skip the remaining part of the body  
  if (i % 2 == 0) continue;  
  alert(i); // 1, then 3, 5, 7, 9  
}
```

Task

Repeat until the input is correct

Write a loop which prompts for a number greater than 100. If the visitor enters another number – ask them to input again.

The loop must ask for a number until either the visitor enters a number greater than 100 or cancels the input/enters an empty line.

Task Solution

Repeat until the input is correct

Write a loop which prompts for a number greater than 100. If the visitor enters another number – ask them to input again.

The loop must ask for a number until either the visitor enters a number greater than 100 or cancels the input/enters an empty line.

```
let num;  
  
do {  
    num = prompt("Enter a number greater than 100?", 0);  
} while (num <= 100 && num);
```

Task

Output prime numbers

An integer number greater than 1 is called a prime if it cannot be divided without a remainder by anything except 1 and itself.

In other words, $n > 1$ is a prime if it can't be evenly divided by anything except 1 and n .

For example, 5 is a prime, because it cannot be divided without a remainder by 2, 3 and 4.

Write the code which outputs prime numbers in the interval from 2 to n .

For $n = 10$ the result will be 2,3,5,7.

P.S. The code should work for any n , not be hard-tuned for any fixed value.

Task Solution

Output prime numbers

```
let n = 10;

nextPrime:
for (let i = 2; i <= n; i++) { // for each i...

    for (let j = 2; j < i; j++) { // look for a divisor..
        if (i % j == 0) continue nextPrime; // not a prime, go next i
    }

    alert( i ); // a prime
}
```

Task

Task Description

Create a simple **Student Management System** using JavaScript that allows the user to enter student information, store grades, and display a full report.

Main Requirements

1 Collect Student Information

Ask the user to enter:

Student Name

Student Age

2 Collect Student Grades

Ask the user to enter grades repeatedly

Stop asking when the user types **"stop"**

Store all grades inside an **array**

Validation:

Grade must be a number

Grade must be between 0 and 100

3 Store Data in Object

Create a student object that contains:

Name

Age

Grades Array

4 Calculate Statistics

Calculate:

Average grade

Highest grade

Lowest grade

5 Display Student Report

Display a final report including:

Student Name

Student Age

All Grades

Average Grade

Student Status

Status Rules

Average $\geq 60 \rightarrow$ Passed

Average $< 60 \rightarrow$ Failed

Bonus Task

Bonus 1 — Input Validation

If user enters:

Text instead of number

Number outside 0 → 100

Show error message

Ask again

Bonus 2 — Minimum Grades Check

Do not allow finishing input unless:

User entered at least 3 grades

Bonus 3 — Grade Classification

Add grade level:

90+ → Excellent

80+ → Very Good

70+ → Good

60+ → Pass

Less than 60 → Fail

Bonus 4 — Continue System

After showing report:

Ask user:

Do you want to enter another student?

Bonus 5 — Show Total Number of Grades

Display:

Total Grades Entered: 5

Last Bonus Store multiple students in an array and

display:

All student names

Average for each student

Functions

Quite often we need to perform a similar action in many places of the script.

For example, we need to show a nice-looking message when a visitor logs in, logs out and maybe somewhere else.

Functions are the main “building blocks” of the program. They allow the code to be called many times without repetition.

We’ve already seen examples of built-in functions, like `alert(message)`, `prompt(message, default)` and `confirm(question)`. But we can create functions of our own as well.

Functions

Function Declaration

To create a function we can use a function declaration.

It looks like this:

```
function showMessage()
```

```
{alert( 'Hello everyone!' );}
```

```
showMessage(); // and this is how call function
```

The function keyword goes first, then goes the name of the function, then a list of parameters between the parentheses (comma-separated, empty in the example above, we'll see examples later) and finally the code of the function, also named “the function body”, between curly braces.

```
function name(parameter1, parameter2, ... parameterN) { // body }
```

Functions

Local variables

A variable declared inside a function is only visible inside that function.

For example:

```
function showMessage() {
```

```
  let message = "Hello, I'm JavaScript!"; // local variable
```

```
  alert( message );}
```

```
showMessage(); // Hello, I'm JavaScript!
```

```
alert( message ); // <-- Error! The variable is local to the function
```

Functions

Outer variables

A function can access an outer variable as well, for example:

```
let userName = 'John';  
  
function showMessage() {  
  
  userName = "Bob"; // (1) changed the outer variable  
  
  let message = 'Hello, ' + userName;  
  
  alert(message);}  
  
alert( userName ); // John before the function call  
  
showMessage();  
  
alert( userName ); // Bob, the value was modified by the function
```

Functions

Parameters

We can pass arbitrary data to functions using parameters.

In the example below, the function has two parameters: from and text

```
function showMessage(from, text) { // parameters: from, text
```

```
  alert(from + ': ' + text);}
```

```
showMessage('Ann', 'Hello!'); // Ann: Hello!
```

```
showMessage('Ann', "What's up?"); // Ann: What's up?
```

Functions

Default values

If a function is called, but an argument is not provided, then the corresponding value becomes undefined.

For instance, the aforementioned function `showMessage(from, text)` can be called with a single argument:

```
function showMessage(from, text = "no text given")
```

```
{alert( from + ": " + text );}
```

```
showMessage("Ann"); // Ann: no text given
```

```
showMessage("Ann", undefined); // Ann: no text given
```

Functions

Returning a value

A function can return a value back into the calling code as the result.

The simplest example would be a function that sums two values:

```
function sum(a, b)
```

```
{return a + b;}
```

```
let result = sum(1, 2);
```

```
alert( result ); // 3
```

Functions

The directive return can be in any place of the function. When the execution reaches it, the function stops, and the value is returned to the calling code (assigned to result above).

There may be many occurrences of return in a single function. For instance:

```
function checkAge(age) {  
  
  if (age >= 18) { return true;}  
  
  else { return confirm('Do you have permission from your parents?');}  
  
}  
  
let age = prompt('How old are you?', 18);  
  
if ( checkAge(age) ) {alert( 'Access granted' );}  
  
  else {alert( 'Access denied' );}
```

Functions

Naming a function

Functions are actions. So their name is usually a verb. It should be brief, as accurate as possible and describe what the function does, so that someone reading the code gets an indication of what the function does.

It is a widespread practice to start a function with a verbal prefix which vaguely describes the action.

For instance, functions that start with "show" usually show something.

Function starting with...

- "get..." – return a value,
- "calc..." – calculate something,
- "create..." – create something,
- "check..." – check something and return a boolean

Task

Function `pow(x,n)`

Write a function `pow(x,n)` that returns `x` in power `n`. Or, in other words, multiplies `x` by itself `n` times and returns the result.

$$\text{pow}(3, 2) = 3 * 3 = 9$$

$$\text{pow}(3, 3) = 3 * 3 * 3 = 27$$

$$\text{pow}(1, 100) = 1 * 1 * \dots * 1 = 1$$

Task Solution

Function pow(x,n)

```
function pow(x, n) {  
    let result = x;  
  
    for (let i = 1; i < n; i++) {  
        result *= x;  
    }  
  
    return result;  
}  
  
let x = prompt("x?", '');  
let n = prompt("n?", '');  
  
if (n < 1) {  
    alert(`Power ${n} is not supported, use a positive integer`);  
} else {  
    alert( pow(x, n) );  
}
```

Functions

Function expressions

In JavaScript, a function is not a “magical language structure”, but a special kind of value.

The syntax that we used before is called a Function Declaration:

```
function sayHi()  
{alert( "Hello" );}
```

There is another syntax for creating a function that is called a Function Expression.

It allows us to create a new function in the middle of any expression.

For example:

```
let sayHi = function() {alert( "Hello" );};
```

Functions

Callback functions

Let's look at more examples of passing functions as values and using function expressions.

We'll write a function `ask(question, yes, no)` with three parameters:

question

Text of the question

yes

Function to run if the answer is "Yes"

no

Function to run if the answer is "No"

Functions

Callback functions

The function should ask the question and, depending on the user's answer, call yes() or no():

```
function ask(question, yes, no)
```

```
{if (confirm(question)) yes()
```

```
else no();}
```

```
function showOk()
```

```
{alert( "You agreed." );}
```

```
function showCancel()
```

```
{alert( "You canceled the execution." );}
```

```
// usage: functions showOk, showCancel are passed as arguments to ask
```

```
ask("Do you agree?", showOk, showCancel);
```

Functions

Function Expression vs Function Declaration

Let's formulate the key differences between Function Declarations and Expressions.

First, the syntax: how to differentiate between them in the code.

- Function Declaration: a function, declared as a separate statement, in the main code flow:

// Function Declaration

```
function sum(a, b) {return a + b;}
```

- Function Expression: a function, created inside an expression or inside another syntax construct. Here, the function is created on the right side of the “assignment expression” =:

// Function Expression

```
let sum = function(a, b) {return a + b;};
```

Functions

A Function Declaration can be called earlier than it is defined.

For example, a global Function Declaration is visible in the whole script, no matter where it is. That's due to internal algorithms. When JavaScript prepares to run the script, it first looks for global Function Declarations in it and creates the functions. We can think of it as an “initialization stage”.

And after all Function Declarations are processed, the code is executed. So it has access to these functions.

For example, this works:

```
sayHi("John"); // Hello, John
```

```
function sayHi(name) {alert( `Hello, ${name}` );}
```

The Function Declaration sayHi is created when JavaScript is preparing to start the script and is visible everywhere in it.

Functions

...If it were a Function Expression, then it wouldn't work:

```
sayHi("John"); // error!
```

```
let sayHi = function(name)
```

```
{ // (*) no magic any more
```

```
alert( `Hello, ${name}` );};
```

Functions

Arrow functions, the basics

There's another very simple and concise syntax for creating functions, that's often better than Function Expressions.

It's called "arrow functions", because it looks like this:

```
let func = (arg1, arg2, ..., argN) => expression;
```

example:

```
let sum = (a, b) => a + b;
```

/* This arrow function is a shorter form of:

```
let sum = function(a, b) {
```

```
  return a + b;
```

```
};
```

```
*/alert( sum(1, 2) ); // 3
```

Functions

Multiline arrow functions

The arrow functions that we've seen so far were very simple. They took arguments from the left of `=>`, evaluated and returned the right-side expression with them.

Sometimes we need a more complex function, with multiple expressions and statements. In that case, we can enclose them in curly braces. The major difference is that curly braces require a `return` within them to return a value (just like a regular function does).

Like this:

```
let sum = (a, b) => {  
  let result = a + b;  
  return result;  
};  
alert( sum(1, 2) ); // 3
```

Thank you