JS Documentation

Introduction

JavaScript is a cross-platform, object-oriented scripting language. It is a small and lightweight language. Inside a host environment (for example, a web browser), JavaScript can be connected to the objects of its environment to provide programmatic control over them.

JavaScript contains a standard library of objects, such as Array, Date, and Math, and a core set of language elements such as operators, control structures, and statements. Core JavaScript can be extended for a variety of purposes by supplementing it with additional objects; for example:

- Client-side JavaScript extends the core language by supplying objects to control a browser and its Document Object Model (DOM). For example, client-side extensions allow an application to place elements on an HTML form and respond to user events such as mouse clicks, form input, and page navigation.
- Server-side JavaScript extends the core language by supplying objects relevant to running JavaScript on a server. For example, server-side extensions allow an application to communicate with a database, provide continuity of information from one invocation to another of the application, or perform file manipulations on a server.

What you should already know

JS Documentation

This guide assumes you have the following basic background:

- A general understanding of the Internet and the World Wide Web (WWW).
- Good working knowledge of HyperText Markup Language (HTML).
- Some programming experience. If you are new to programming, try one of the tutorials linked on the main page about JavaScript.

JavaScript and Java

JavaScript and Java are similar in some ways but fundamentally different in some others. The JavaScript language resembles Java but does not have Java's static typing and strong type checking. JavaScript follows most Java expression syntax, naming conventions and basic control-flow constructs which was the reason why it was renamed from LiveScript to JavaScript.

In contrast to Java's compile-time system of classes built by declarations, JavaScript supports a runtime system based on a small number of data types representing numeric, Boolean, and string values. JavaScript has a prototype-based object model instead of the more common class-based object model. The prototype-based model provides dynamic inheritance; that is, what is inherited can vary for individual objects. JavaScript also supports functions without any special declarative requirements. Functions can be properties of objects, executing as loosely typed methods.

JavaScript is a very free-form language compared to Java. You do not have to declare all variables, classes, and methods. You do not have to be concerned with whether methods are public, private, or protected, and you do not have to

JS Documentation

implement interfaces. Variables, parameters, and function return types are not explicitly typed.

Hello world

To get started with writing JavaScript, open the Scratchpad and write your first "Hello world" JavaScript code:

```
function greetMe(yourName) {
alert("Hello " + yourName); }
greetMe("World");
```

Select the code in the pad and hit Ctrl+R to watch it unfold in your browser!

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 must start with a letter, underscore (_), or dollar sign ($); subsequent characters can also be digits (0-9). Because JavaScript is case sensitive, letters include the characters "A" through "Z" (uppercase) and the characters "a" through "z" (lowercase).

You can use ISO 8859-1 or Unicode letters such as å and ü in identifiers. You can also use the Unicode escape sequences as characters in identifiers. Some examples of legal names are Number_hits, temp99, and _name.

Declaring variables

You can declare a variable in three ways:

With the keyword var. For example,

JS Documentation

```
var x = 42.
```

This syntax can be used to declare both local and global variables.

By simply assigning it a value. For example,

```
x = 42.
```

This always declares a global variable. It generates a strict JavaScript warning. You shouldn't use this variant.

With the keyword let. For example,

```
let y = 13.
```

This syntax can be used to declare a block scope local variable. See Variable scope below.

Variable scope

When you declare a variable outside of any function, it is called a global variable, because it is available to any other code in the current document. When you declare a variable within a function, it is called a local variable, because it is available only within that function.

JavaScript before ECMAScript 2015 does not have block statement scope; rather, a variable declared within a block is local to the function (or global scope) that the block resides within. For example the following code will log 5, because the scope of x is the function (or global context) within which x is declared, not the block, which in this case is an if statement.

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

JS Documentation

This behavior changes, when using the let declaration introduced in ECMAScript 2015.

```
if (true) { let y = 5; } console.log(y);
// ReferenceError: y is
not defined
```

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.

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.

Constants

You can create a read-only, named constant with the const keyword. The syntax of a constant identifier is the same as for a variable identifier: it must start with a letter, underscore or dollar sign and can contain alphabetic, numeric, or underscore characters.

```
const PI = 3.14;
```

A constant cannot change value through assignment or be re-declared while the script is running. It has to be initialized to a value.

The scope rules for constants are the same as those for let block scope variables. If the const

JS Documentation

keyword is omitted, the identifier is assumed to represent a variable.

You cannot declare a constant with the same name as a function or variable in the same scope. For example:

```
// THIS WILL CAUSE AN ERROR function f()
{}; const f = 5; // THIS
WILL CAUSE AN ERROR ALSO function f() {
const g = 5; var g;
//statements }
```

However, object attributes are not protected, so the following statement is executed without problems.

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

Data types

The latest ECMAScript standard defines seven data types:

- Six data types that are primitives:
    - Boolean. true and false.
    - null. A special keyword denoting a null value. Because JavaScript is case-sensitive, null is not the same as Null, NULL, or any other variant.
    - undefined. A top-level property whose value is undefined.
    - Number. 42 or 3.14159.
    - String. "Howdy"

    - Symbol (new in ECMAScript 2015). A data type whose instances are unique and immutable.
- and Object

Although these data types are a relatively small amount, they enable you to perform useful functions with your applications. Objects and functions are the other fundamental elements in the language. You can think of objects as named containers for values, and functions as procedures that your application can perform.

if...else statement

Use the if statement to execute a statement if a logical condition is true. Use the optional else clause to execute a statement if the condition is false. An if statement looks as follows:

```
if (condition) { statement_1; } else {
statement_2; }
```

condition can be any expression that evaluates to true or false. See Boolean for an explanation of what evaluates to true and false. If condition evaluates to true, statement_1 is executed; otherwise, statement_2 is executed. statement_1 and statement_2 can be any statement, including further nested if statements.

You may also compound the statements using else if to have multiple conditions tested in sequence, as follows:

```
if (condition_1) { statement_1; } else
if (condition_2) {
statement_2; } else if (condition_n) {
statement_n; } else {
statement_last; }
```

In the case of multiple conditions only the first logical condition which evaluates to true will be executed. To execute multiple statements, group them within a block statement ({ ... }) . In general, it's good practice to always use block statements, especially when nesting if statements:

```
if (condition) {
statement_1_runs_if_condition_is_true;
statement_2_runs_if_condition_is_true; }
else {
statement_3_runs_if_condition_is_false;
statement_4_runs_if_condition_is_false;
}
```

It is advisable to not use simple assignments in a conditional expression, because the assignment can be confused with equality when glancing over the code. For example, do not use the following code:

```
if (x = y) { /* statements here */ }
```

If you need to use an assignment in a conditional expression, a common practice is to put additional parentheses around the assignment. For example:

```
if ((x = y)) { /* statements here */ }
```

while statement

A while statement executes its statements as long as a specified condition evaluates to true. A while statement looks as follows:

```
while (condition) statement
```

If the condition becomes false, statement within the loop stops executing and control passes to the

JS Documentation

statement following the loop.

The condition test occurs before statement in the loop is executed. If the condition returns true, statement is executed and the condition is tested again. If the condition returns false, execution stops and control is passed to the statement following while.

To execute multiple statements, use a block statement ({ ... }) to group those statements.

Example:

The following while loop iterates as long as n is less than three:

```
var n = 0; var x = 0; while (n < 3) {
n++; x += n; }
```

With each iteration, the loop increments n and adds that value to x. Therefore, x and n take on the following values:

- After the first pass: n = 1 and x = 1
- After the second pass: n = 2 and x = 3
- After the third pass: n = 3 and x = 6

After completing the third pass, the condition n < 3 is no longer true, so the loop terminates.

Function declarations

A function definition (also called a function declaration, or function statement) consists of the function keyword, followed by:

- The name of the function.
- A list of arguments to the function, enclosed in parentheses and separated by commas.

JS Documentation

- The JavaScript statements that define the function, enclosed in curly brackets, { }.

For example, the following code defines a simple function named square:

```
function square(number) { return number
* number; }
```

The function square takes one argument, called number. The function consists of one statement that says to return the argument of the function (that is, number) multiplied by itself. The return statement specifies the value returned by the function.

```
return number * number;
```

Primitive parameters (such as a number) are passed to functions by value; the value is passed to the function, but if the function changes the value of the parameter, this change is not reflected globally or in the calling function.

Reference

- All the documentation in this page is taken from MDN