JavaScript Arrays (exp Array.html)

JavaScript arrays are written with square brackets.

Array items are separated by commas.

The following code declares (creates) an array called `cars`, containing three items (car names):

Example

```
var cars = ["Saab", "Volvo", "BMW"];
```

Array indexes are zero-based, which means the first item is [0], second is [1], and so on.

You will learn more about arrays later in this tutorial.

JavaScript Objects (ex Object.html)

JavaScript objects are written with curly braces `{}`.

Object properties are written as name:value pairs, separated by commas.

Example

```
var person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
```

The object (person) in the example above has 4 properties: firstName, lastName, age, and eyeColor.

You will learn more about objects later in this tutorial.

The typeof Operator

You can use the JavaScript `typeof` operator to find the type of a JavaScript variable.

The `typeof` operator returns the type of a variable or an expression:

Example

```
typeof ""              // Returns "string"
typeof "John"          // Returns "string"
typeof "John Doe"      // Returns "string"
```

Example

```
typeof 0               // Returns "number"
typeof 314             // Returns "number"
typeof 3.14            // Returns "number"
typeof (3)             // Returns "number"
typeof (3 + 4)         // Returns "number"
```

Undefined

In JavaScript, a variable without a value, has the value `undefined`. The type is also `undefined`.

Example

```
var car;    // Value is undefined, type is undefined
```

Any variable can be emptied, by setting the value to `undefined`. The type will also be `undefined`.

Example

```
car = undefined;    // Value is undefined, type is undefined
```

Empty Values

An empty value has nothing to do with `undefined`.

An empty string has both a legal value and a type.

Example

```
var car = "";    // The value is "", the typeof is "string"
```

Null

In JavaScript `null` is "nothing". It is supposed to be something that doesn't exist.

Unfortunately, in JavaScript, the data type of `null` is an object.

You can consider it a bug in JavaScript that `typeof null` is an object. It should be `null`.

You can empty an object by setting it to `null`:

Example

```
var person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
person = null;    // Now value is null, but type is still an
object
```

You can also empty an object by setting it to `undefined`:

Example

```
var person = {firstName:"John", lastName:"Doe", age:50,
eyeColor:"blue"};
person = undefined;   // Now both value and type is undefined
```

Difference Between Undefined and Null

`undefined` and `null` are equal in value but different in type:

```
typeof undefined           // undefined
typeof null                // object

null === undefined         // false
null == undefined          // true
```

Primitive Data

A primitive data value is a single simple data value with no additional properties and methods.

The `typeof` operator can return one of these primitive types:

- `string`
- `number`
- `boolean`
- `undefined`

Example

```
typeof "John"              // Returns "string"
typeof 3.14                // Returns "number"
typeof true                // Returns "boolean"
typeof false               // Returns "boolean"
typeof x                   // Returns "undefined" (if x has no
value)
```

Complex Data

The `typeof` operator can return one of two complex types:

- `function`
- `object`

The `typeof` operator returns "object" for objects, arrays, and null.

The `typeof` operator does not return "object" for functions.

Example

```
typeof {name:'John', age:34} // Returns "object"
typeof [1,2,3,4]             // Returns "object" (not "array",)
typeof null                  // Returns "object"
typeof function myFunc(){}   // Returns "function"
```

Functions and Scope

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)` . But we can create functions of our own as well.

Function Declaration

To create a function we can use a *function declaration*.

It looks like this:

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

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) and finally the code of the function, also named “the function body”, between curly braces.

```
function name(parameters) {
  ...body...
}
```

Our new function can be called by its name: `showMessage()`.

For instance:

```

```

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

showMessage();
showMessage();
```

The call `showMessage()` executes the code of the function. Here we will see the message two times.

This example clearly demonstrates one of the main purposes of functions: to avoid code duplication.

If we ever need to change the message or the way it is shown, it's enough to modify the code in one place: the function which outputs it.

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
```

Outer variables

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

```
let userName = 'John';

function showMessage() {
```

```
  let message = 'Hello, ' + userName;
  alert(message);
}

showMessage(); // Hello, John
```

The function has full access to the outer variable. It can modify it as well.

For instance:

```
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
```

The outer variable is only used if there's no local one.

If a same-named variable is declared inside the function then it *shadows* the outer one. For instance, in the code below the function uses the local `userName`. The outer one is ignored:

```

let userName = 'John';

function showMessage() {
  let userName = "Bob"; // declare a local variable

  let message = 'Hello, ' + userName; // Bob
  alert(message);
}

// the function will create and use its own userName
showMessage();

alert( userName ); // John, unchanged, the function did not access the
outer variable
```

Global variables

Variables declared outside of any function, such as the outer `userName` in the code above, are called *global*.

Global variables are visible from any function (unless shadowed by locals).

It's a good practice to minimize the use of global variables. Modern code has few or no globals. Most variables reside in their functions. Sometimes though, they can be useful to store project-level data.

Parameters

We can pass arbitrary data to functions using parameters (also called *function arguments*) .

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

```
function showMessage(from, text) { // arguments: from, text
  alert(from + ': ' + text);
}

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

When the function is called in lines `(*)` and `(**)`, the given values are copied to local variables `from` and `text`. Then the function uses them.

Here's one more example: we have a variable `from` and pass it to the function. Please note: the function changes `from`, but the change is not seen outside, because a function always gets a copy of the value:

```

function showMessage(from, text) {

  from = '*' + from + '*'; // make "from" look nicer

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

let from = "Ann";

showMessage(from, "Hello"); // *Ann*: Hello

// the value of "from" is the same, the function modified a local copy
alert( from ); // Ann
```

Default values

If a parameter is not provided, then its value becomes `undefined`.

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

```
showMessage("Ann");
```

That's not an error. Such a call would output `"Ann: undefined"`. There's no `text`, so it's assumed that `text === undefined`.

If we want to use a "default" `text` in this case, then we can specify it after `=`:

```
function showMessage(from, text = "no text given") {
  alert( from + ": " + text );
}

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

Now if the `text` parameter is not passed, it will get the value `"no text given"`

Here `"no text given"` is a string, but it can be a more complex expression, which is only evaluated and assigned if the parameter is missing. So, this is also possible:

```
function showMessage(from, text = anotherFunction()) {
  // anotherFunction() only executed if no text given
  // its result becomes the value of text
}
```

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
```

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' );
}
```
It is possible to use `return` without a value. That causes the function to exit immediately.

For example:

```
function showMovie(age) {
  if ( !checkAge(age) ) {
    return;
  }

  alert( "Showing you the movie" ); // (*)
  // ...
}
```
In the code above, if `checkAge(age)` returns `false`, then `showMovie` won't proceed to the `alert`.

A function with an empty `return` or without it returns `undefined`

If a function does not return a value, it is the same as if it returns `undefined`:

```
function doNothing() { /* empty */ }

alert( doNothing() === undefined ); // true
```
An empty `return` is also the same as `return undefined`:

```
function doNothing() {
  return;
}

alert( doNothing() === undefined ); // true
```
Never add a newline between `return` and the value

For a long expression in `return`, it might be tempting to put it on a separate line, like this:

```
return
 (some + long + expression + or + whatever * f(a) + f(b))
```
That doesn't work, because JavaScript assumes a semicolon after `return`. That'll work the same as:

```
     return;
 (some + long + expression + or + whatever * f(a) + f(b))
```
So, it effectively becomes an empty return.

If we want the returned expression to wrap across multiple lines, we should start it at the same line as `return`. Or at least put the opening parentheses there as follows:

```
return (
  some + long + expression
  + or +
  whatever * f(a) + f(b)
  )
```

And it will work just as we expect it to.

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. There must be an agreement within the team on the meaning of the prefixes.

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, etc.

Examples of such names:

```
showMessage(..)      // shows a message
getAge(..)            // returns the age (gets it somehow)
calcSum(..)           // calculates a sum and returns the result
createForm(..)       // creates a form (and usually returns it)
checkPermission(..) // checks a permission, returns true/false
```

With prefixes in place, a glance at a function name gives an understanding what kind of work it does and what kind of value it returns.

One function – one action

A function should do exactly what is suggested by its name, no more.

Two independent actions usually deserve two functions, even if they are usually called together (in that case we can make a 3rd function that calls those two).

A few examples of breaking this rule:

- `getAge` – would be bad if it shows an `alert` with the age (should only get).
- `createForm` – would be bad if it modifies the document, adding a form to it (should only create it and return).

- `checkPermission` – would be bad if it displays the `access granted/denied` message (should only perform the check and return the result).

These examples assume common meanings of prefixes. You and your team are free to agree on other meanings, but usually they're not much different. In any case, you should have a firm understanding of what a prefix means, what a prefixed function can and cannot do. All same-prefixed functions should obey the rules. And the team should share the knowledge.

Ultrashort function names

Functions that are used *very often* sometimes have ultrashort names.

For example, the jQuery framework defines a function with `$`. The Lodash library has its core function named `_`.

These are exceptions. Generally functions names should be concise and descriptive.

Functions should be short and do exactly one thing. If that thing is big, maybe it's worth it to split the function into a few smaller functions. Sometimes following this rule may not be that easy, but it's definitely a good thing.

A separate function is not only easier to test and debug – its very existence is a great comment!

For instance, compare the two functions `showPrimes(n)` below. Each one outputs prime numbers up to `n`.

The first variant uses a label:

```
function showPrimes(n) {
  nextPrime: for (let i = 2; i < n; i++) {

    for (let j = 2; j < i; j++) {
      if (i % j == 0) continue nextPrime;
    }

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

The second variant uses an additional function `isPrime(n)` to test for primality:

```
                              function showPrimes(n) {

  for (let i = 2; i < n; i++) {
```

```
    if (!isPrime(i)) continue;

    alert(i);  // a prime
  }
}

function isPrime(n) {
  for (let i = 2; i < n; i++) {
    if ( n % i == 0) return false;
  }
  return true;
}
```

The second variant is easier to understand, isn't it? Instead of the code piece we see a name of the action (`isPrime`). Sometimes people refer to such code as *self-describing*.

So, functions can be created even if we don't intend to reuse them. They structure the code and make it readable.

Method examples

For a start, let's teach the `user` to say hello:

```
let user = {
  name: "John",
  age: 30
};

user.sayHi = function() {
  alert("Hello!");
};

user.sayHi(); // Hello!
```

Here we've just used a Function Expression to create the function and assign it to the property `user.sayHi` of the object.

Then we can call it. The user can now speak!

A function that is the property of an object is called its *method*.

So, here we've got a method `sayHi` of the object `user`.

Of course, we could use a pre-declared function as a method, like this:

```
let user = {
  // ...
};

// first, declare
function sayHi() {
  alert("Hello!");
};

// then add as a method
user.sayHi = sayHi;

user.sayHi(); // Hello!
```

Object-oriented programming

When we write our code using objects to represent entities, that's called object-oriented programming, in short: "OOP".

OOP is a big thing, an interesting science of its own. How to choose the right entities? How to organize the interaction between them? That's architecture, and there are great books on that topic, like "Design Patterns: Elements of Reusable Object-Oriented Software" by E. Gamma, R. Helm, R. Johnson, J. Vissides or "Object-Oriented Analysis and Design with Applications" by G. Booch, and more.

Method shorthand

There exists a shorter syntax for methods in an object literal:

```
// these objects do the same

user = {
  sayHi: function() {
    alert("Hello");
  }
};

// method shorthand looks better, right?
user = {
  sayHi() { // same as "sayHi: function()"
    alert("Hello");
  }
};
```

As demonstrated, we can omit `"function"` and just write `sayHi()`.

To tell the truth, the notations are not fully identical. There are subtle differences related to object inheritance (to be covered later), but for now they do not matter. In almost all cases the shorter syntax is preferred.

“this” in methods

It’s common that an object method needs to access the information stored in the object to do its job.

For instance, the code inside `user.sayHi()` may need the name of the `user`.

To access the object, a method can use the `this` keyword.

The value of `this` is the object “before dot”, the one used to call the method.

For instance:

```
let user = {
  name: "John",
  age: 30,

  sayHi() {
    // "this" is the "current object"
    alert(this.name);
  }

};

user.sayHi(); // John
```

Here during the execution of `user.sayHi()`, the value of `this` will be `user`.