JavaScript Beginner Tutorial

As with the HTML Beginner Tutorial and CSS Beginner Tutorial, the **JavaScript Beginner Tutorial** assumes that you know zip all about JavaScript. The purpose of this guide is to teach the bare essentials — just enough to get started. The JavaScript Intermediate Tutorial and JavaScript Advanced Tutorial go into more depth.

This tutorial is an introduction to programming in general, through the medium of JavaScript. We'll get into interacting with web pages in the Intermediate section.

As thrilling as it is to jump in at the deep end, these waters contain rabid sharks AND angry piranhas. That's right; salt water fish AND fresh water fish all in one swimming pool.

For your own safety, **you should be as comfortable as a cushion with HTML and CSS before proceeding**. Don't say we didn't warn you…

 **Link To Us!** If you've found HTML Dog useful, please consider linking to us.

Contents

Related pages

- Next Page: Making Stuff Happen
- HTML Beginner Tutorial
- JavaScript Intermediate Tutorial

https://htmldog.com/guides/javascript/beginner/

Making Stuff Happen

There's a few ways to try out JavaScript, and when learning it's best to try them out to see what works for you. But first, how does JavaScript relate to HTML and CSS?

HTML, CSS and JavaScript

Mostly, JavaScript runs in your web browser alongside HTML and CSS, and can be added to any web page using a `script` tag. The `script` element can either contain JavaScript directly (**internal**) or link to an external resource via a `src` attribute (**external**).

A browser then runs JavaScript line-by-line, starting at the top of the file or `script` element and finishing at the bottom (unless you tell it to go elsewhere).

Internal

You can just put the JavaScript inside a `script` element:

```
<script>
    alert("Hello, world.");
</script>
```


External

An external JavaScript resource is a **text file** with a **.js** extension, just like an external CSS resource with a **.css** extension.

To add a JavaScript file to your page, you just need to use a script tag with a src attribute pointing to the file. So, if your file was called script.js and sat in the same directory as your HTML file, your `script` element would look like this:

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

You might also come across another way on your view-source travels: inline. This involves event attributes inside HTML tags that look something like `<a href="somewhere.html" onclick="alert('Noooooo!');">Click me</a>`. Just move along and pretend you haven't witnessed this aberration. We really, really, really want to separate our technologies so it's preferable to avoid this approach.

Console

The last way is great for getting instant feedback, and it's recommend if you just want to try a line out quickly.

In a modern browser you'll find some developer tools - often you can right click on a page, then click "inspect element" to bring them up. Find the console and you'll be able to type JavaScript, hit enter and have it run immediately.

Search the net if you can't find your dev tools or console - they're changing in browsers all the time.

Let's go!

To get started, the best way is the internal method. You can modify the contents of the `script` element and refresh the page just like you're used to - quick and easy.

Now it's time to learn some JavaScript. Good luck!

Variables and Data

Storing data so we can use it later is one of the most important things when writing code. Fortunately, JavaScript can do this! If it couldn't, it'd be pretty darn useless.

So let's ask the user (that's you) for their surname (check your driving license, or ask a friend).

By the way, this assumes you've jumped into a browser and are typing into your console.

```
var surname = prompt('Greetings friend, may I enquire as to your surname?');
```

A little box will pop-up, asking (very courteously if I may say) for your surname. Type your surname in and hit 'OK'.

The surname you entered is now saved, and it can be referred to as *surname*. You can get what you entered back out again by typing *surname* into the console. You should see your surname appearing back to you underneath! Exciting!

You've created a *variable*.

 **New Examples Section!** See all of this code stuff in action, and play around with it.

Variables

Think, if you will, of a variable as a shelf with a name so it's easy to get back to. You've named a shelf *surname*.

You can name a variable anything you like, but it's best to name it in a way that tells what you've stored there. For example, *surname* is better than *mything* or *s*.

When you type a variable name into the console you are asking the browser, which looks after the shelves, to go find the shelf and give what's on it to you. This is also known as the variable's *value*. The value can be almost anything - in *surname*, you've stored some letters, known as a *string*. You can also store numbers and a myriad other kinds of data.

So, a variable has a *name* and a *value*.

They are the way we store data, and you'll be using them a lot.

There are two parts to creating a variable; *declaration* and *initialization*. Once it's created, you can *assign* (or set) its value.

Declaration

Declaration is *declaring* a variable to *exist*. To return to the shelf metaphor, it's like picking an empty shelf in a massive warehouse and putting a name on it.

As above, to declare a variable, use the `var` keyword followed by the variable name, like this:

```
var surname;
var age;
```

Notice those semicolons (";")? Almost every line in JavaScript ends in a semicolon - you'll be using them a lot.

Initialization

Initialization is giving a variable its *value* for the *first time*. The value can change later, but it is only initialized once.

You initialize a variable using the equals sign (=). You can read it as "the *value* of the variable on the left should be the data on the right":

```
var name = "Tom";
```

"Tom" is a *string* - a collection of letters. A string is surrounded by single or double quote marks.

```
var age = 20;
```

20 is just a number - and numbers don't go in quotes.

Assignment

As mentioned, you can set a variable's value as many times as you like. It's called *assignment* and it looks very similar to initialization. You again use the equals sign, but there's no need for the `var` keyword because we've already declared the variable.

It's like this, yo:

```
name = "Andy";
age = 43;
```

Only do this if you've declared the variable using the `var` keyword!

Doing Math

Variables can be used to store strings and numbers (amongst other things), but here the focus is on numbers.

How much fruit?

In your console, let's create two variables. One will be the number of apples we have, the second will be the number of pears.

```
var apples = 5, pears = 10;
```

That creates two variables… but there's only one `var` keyword? Yup, it's a shorthand for declaring and initializing multiple variables at the same time. By using the `var` keyword once and separating the new variables with commas you can save yourself some work.

Now, use these two variables to figure out how many pieces of fruit there are in total.

```
var piecesOfFruit = apples + pears;
```

So that's new.

 **New Examples Section!** See all of this code stuff in action, and play around with it.

You're asking the browser to work out the sum on the right before assigning the result to *piecesOfFruit* - and you're not adding the variable names, you're adding the *values of the variables*. The browser knows that, when it sees a variable like this, you want to do something with the value. So it goes and gets each variable's value before doing the sum.

The sum is called *piecesOfFruit*, not *pieces of fruit*. This is because variable names cannot contain spaces! There's a set of rules somewhere about what you can and can't use, but for now just use letters with no spaces.

You may have also noticed that the uppercase "O" on "Of" and "F" on "Fruit". This is a convention, called *camel casing* (cos of the humps), that makes the variable name easier to read. You can call a variable anything you want, but *pIeCEsOFfRuiT* is pretty hard to read, no?

Here's a tip for you: code is read many, many more times that it is written. So it's really, really important to make sure your code is easy to read. Really important.

Splitting the fruit

Let’s say you want to split your fruit between 3 people. How much fruit does each person get?

```
var piecesForEachPerson = piecesOfFruit / 3;
```

Thank goodness we have this JavaScript thing, otherwise you’d need a calculator for that one…

You’re using a forward slash (“ / ”) to indicate division: divide the thing on the left by the thing on the right.

Precedence & operators

For doing sums you can use various symbols: add (“ + ”), subtract (“ - ”), divide (“ / ”) and multiply (“ * ”).

Mathematical symbols are called *operators*; that is, they *operate* on some data. We’ll meet even more operators later on, but you should know that, like on a calculator, the symbols are worked out in a particular order, called *operator precedence*. Things in parentheses - that’s these: “ (” and “) ” - are done first, then multiplication and division, then addition and subtraction.

Here’s an example, stepping through how the browser runs it:

```
(10 + 2) / 2 + 4 * 2
```

The part in brackets is worked out first. So this becomes…

```
12 / 2 + 4 * 2
```

…which works out to…

```
6 + 4 * 2
```

Then things are done in operator precedence order. If multiple sums are of the same precedence then they are evaluated left to right.

The multiplication is higher precedence so is done first, which works out to…

```
6 + 8
```

This is used to work out the final value:

```
14
```

Whew!

Logic

A really important part of programming is being able to compare values in order to make decisions in code. When a comparison is made the outcome is either true or false; a special kind a of data called a *boolean*. This is *logic*.

Like when doing math, there's a set of operators that work on booleans. They are used to compare two values, on the left and right of the operator, to produce a boolean value.

The following comparisons are made in the console. The result we get from each line (shown underneath) will be `true` or `false`; these are booleans.

Equality

To find out when two values are equal, use the *triple equals* operator ("===").

```
15.234 === 15.234
```

```
true
```

We can also determine if two values are not equal using the *triple not equal* operator ("!==").

```
15.234 !== 18.4545
```

```
true
```

It's important to know that strings containing a number and an actual number are *not equal*.

```
'10' === 10
```

```
false
```


Greater than and less than

Comparing two numbers is useful, for example, to determine which of two is larger or smaller. This first example is a comparison of 10 and 5 to see if 10 is larger, using the *greater than* operator (">").

```
10 > 5
```

```
true
```

Next we use the *less than* operator (“<”) to determine if the left value is smaller.

```
20.4 < 20.2
```

```
false
```

That example gives back (or *returns*) false, because 20.4 is not a smaller number than 20.2.

Combined comparisons

Combining a comparison of equality and size can be done with the *greater than or equal to* and *less than or equal to* operators (“>=” and “<=” respectively).

```
10 >= 10
```

```
true
```

```
10 <= 5
```

```
false
```

Conditional

Logic is used to make decisions in code; choosing to run one piece of code or another depending on the comparisons made. This requires use of something called a *conditional*. There are a few different conditionals that you might want to use, but we'll just focus the one used most commonly: `if`.

If

It's very simple: if some logic (the *condition*) is true, run a *block* of code. You can also supply more code to be run if the condition is not true, and supply additional conditions and blocks of code to optionally be run. These forms are called *if-else*, as you'll see below.

The most simple `if` statement looks something like this:

```
if (10 > 5) {
    // Run the code in here
}
```

The code between the braces - "{" and "}" - is called a *block*, and this one is linked to the `if` statement. It's only run if the *conditional* (between the parentheses) is *true*.

10 is indeed greater than 5, so the code in the block would run.

Oh, and the line starting "//"? The double slashes indicate comment, which means that everything after the slashes until the end of the line is ignored by the browser.

If-else

The if-else form of an `if` statement is used to run an alternative piece of code if the conditional is not true. The code in the `if` block below will be ignored, for example - only the code in the `else` block will be run.

```
if (43 < 2) {
    // Run the code in here
} else {
    // Run a different bit of code
}
```

Looping

Loops are a way of repeating the same block of code over and over. They're incredibly useful, and are used, for example, to carry out an action on every item in an array (we will come to arrays later) or in searching.

Two of the most common loops are *while* loops and *for* loops. They combine a conditional and a block, running the block over and over until the logic of the conditional is no longer true, or until you force them to stop.

While

A `while` loop repeats a block of code *while* a *condition is true*. Like an `if` statement, the condition is found in parentheses.

```
var i = 1;
while (i < 10) {
    alert(i);
    i = i + 1;
}
// i is now 10
```

After looping has finished the code carries on running from just after the closing brace ("}") of the loop's block.

For

A `for` loop is similar to an `if` statement, but it combines three semicolon-separated pieces information between the parentheses: *initialization*, *condition* and a final *expression*.

The *initialization* part is for creating a variable to let you track how far through the loop you are - like *i* in the `while` example; the *condition* is where the looping *logic* goes - the same as the *condition* in the while example; and the final *expression* is run at the *end of every loop*.

```
for (var i = 1; i < 10; i++) {
    alert(i);
}
```

This gives us alert boxes containing the numbers 1 to 10 in order.

By the way, `i++` is equivalent to `i = i + 1`.

Functions

Functions are reusable blocks of code that carry out a specific task. To *execute* the code in a function you *call* it. A function can be passed *arguments* to use, and a function may *return* a value to whatever called it.

You can save functions as the value of a variable, and you call a function using this variable and a pair of parentheses. This is also called *invoking* the function.

```
doSomething();
findAnInterestingThing();
```

To create a function, use the `function` keyword. You then list the arguments in parentheses, and then supply a block that contains the function's code. Here's a function that adds two numbers:

```
var add = function (a, b) {
    return a + b;
};
```

 **New Examples Section!** See all of this code stuff in action, and play around with it.

a and *b* are the function's parameters, and the value it returns is signified by the `return` keyword. The `return` keyword also stops execution of the code in the function; nothing after it will be run.

```
var result = add(1, 2); // result is now 3
```

This calls add with the arguments *1* and *2*, which, inside add, will be saved in the variables *a* and *b*.

Objects

JavaScript objects are like a real life objects; they have properties and abilities. A JavaScript object is, in that sense, a collection of named *properties* and *methods* - a function. An object can be stored in a variable, and the properties and methods accessed using the *dot syntax*.

A human, for example, has a name and an age, and could talk, move or learn JavaScript. Name and age are *properties* of the human, and are essentially pieces of data. Talking, moving and learning are more like functions - there's some complex behavior involved. When a JavaScript object has such an ability, it is called a *method*.

Variables can hold objects, and creating an object is done using a special syntax signified by braces:

```
var jedi = {
    name: "Yoda",
    age: 899,
    talk: function () { alert("another... Sky... walk..."); }
};
```

The Jedi's *name* and *age* are properties - they are essentially variables within the object and can store anything a variable can. *talk* is a property that holds a function - a method.

You can get data back out of an object using the *dot syntax*:

```
jedi.name;
```

```
Yoda
```

```
jedi.age;
```

```
899
```

```
jedi.talk();
```

```
//produces an alert box
```

 **New Examples Section!** See all of this code stuff in action, and play around with it.

You can also reassign properties of an object:

```
jedi.name = "Mace Windu";
```

And add new ones on the fly:

```
jedi.lightsaber = "purple";
```

Properties can be any kind of data including objects and arrays. Adding an object as a property of another object creates a *nested object*:

```
var person = {
    age: 122
};

person.name = {
    first: "Jeanne",
    last: "Calment"
};
```

Creating an empty object and adding properties and methods to it is possible too:

```
var dog = {};

dog.bark = function () { alert("Woof!"); };
```

Arrays

Arrays are lists of any kind of data, including other arrays. Each item in the array has an **index** — a number — which can be used to retrieve an **element** from the array.

The indices start at 0; that is, the first element in the array has the index *0*, and subsequent elements have incrementally increasing indices, so the last element in the array has an index one less than the length of the array.

In JavaScript, you create an array using the *array-literal syntax*:

```
var emptyArray = [];
var shoppingList = ['Milk', 'Bread', 'Beans'];
```

You retrieve a specific element from an array using *square bracket* syntax:

```
shoppingList[0];
```

```
Milk
```

It's also possible to set the value at a particular index, again using the square bracket syntax:

```
shoppingList[1] = 'Cookies';
```

```
// shoppingList is now ['Milk', 'Cookies', 'Beans']
```


You can find the number of elements in the array using its `length` property:

```
shoppingList.length;
```

```
3
```

You can use `push` and `pop` methods to add and remove elements from the end of the array:

```
shoppingList.push('A new car');
```

```
// shoppingList is now ['Milk', 'Bread', 'Beans', 'A new car']
```

```
shoppingList.pop();


// shoppingList is back to ['Milk', 'Bread', 'Beans']
```

Here’s an example that creates, pushes, pops and iterates over an array, passing each name to a function called *helloFrom*. *helloFrom* returns a string with a greeting: “Hello from ” and then the person’s name. After the pushing and popping, the final list of people is: “Tom”, “Yoda”, “Ron” and “Bob”.

```
var helloFrom = function (personName) {
    return "Hello from " + personName;
}

var people = ['Tom', 'Yoda', 'Ron'];

people.push('Bob');
people.push('Dr Evil');

people.pop();

for (var i=0; i < people.length; i++) {
    var greeting = helloFrom(people[i]);
    alert(greeting);
}
```