Console `tool`

Your browser comes with a **developer console** that allows you to type JavaScript directly in this webpage.

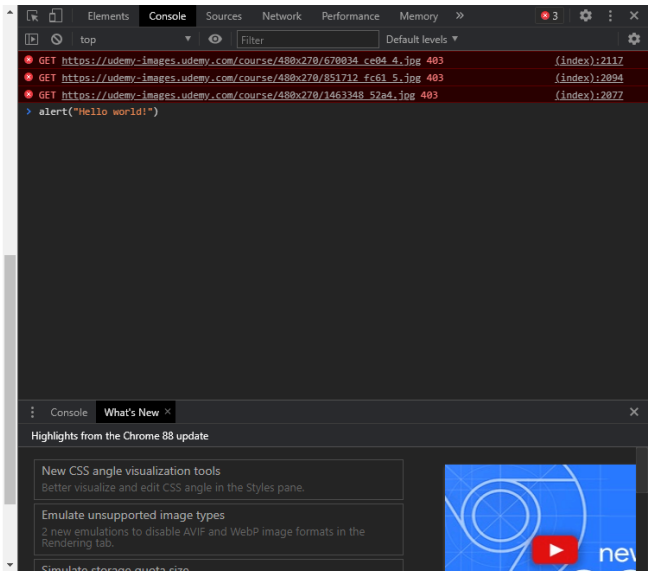
Since you're using **Google Chrome**, open the JavaScript console with `Ctrl` + `Shift` + `J` or `F12`

Let's see if it works!

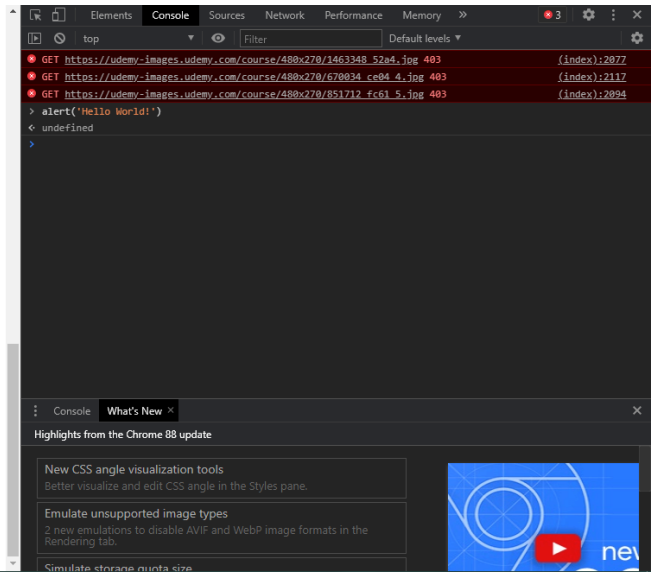
In the console, type (or paste) the following, and press `Enter`

```
alert('Hello World!')
```

Copy to clipboard



Copy to clipboard



Great job!

You've just used `alert()`, a native **JavaScript function** that comes with every browser.

But what have you done exactly?

Functions concept

You've **called** the `alert()` *function* with the `'Hello World!'` *argument*.

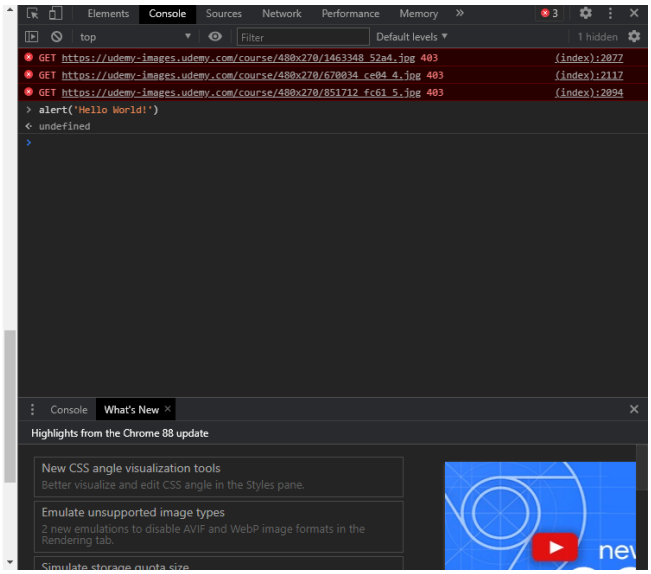
You have basically done 4 things:

1	you typed the **name** of the function	`alert('Hello World!')`
2	you opened a **parenthesis**	`alert('Hello World!')`
3	you typed an **argument**	`alert('Hello World!')`
4	you **closed** the parenthesis	`alert('Hello World!')`

Sometimes, there's no argument.
Sometimes, there's multiple arguments.
Most of them are required, but some of them can be optional.

In this case, `alert()` requires only one argument.

But what type of argument is that?



Strings type

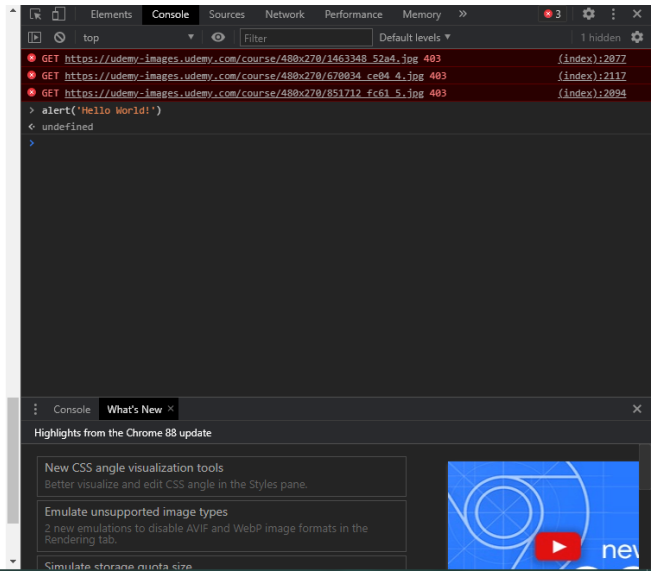
When you handle text, you're using **strings**, which are a series of *characters*. In our case, we used a series of **12 characters**: `Hello World!` . This includes all lowercase and uppercase letters, the space, and the exclamation point.

To define where the string *starts* and where it *ends*, you need to wrap it in **quotes** (either single `'` or double `"`).

When you defined the `'Hello World!'` string:

1. you typed a single **quote**
2. you typed the **12 characters**
3. you typed another single **quote**

What if you wanted to deal with numbers instead?



Numbers type

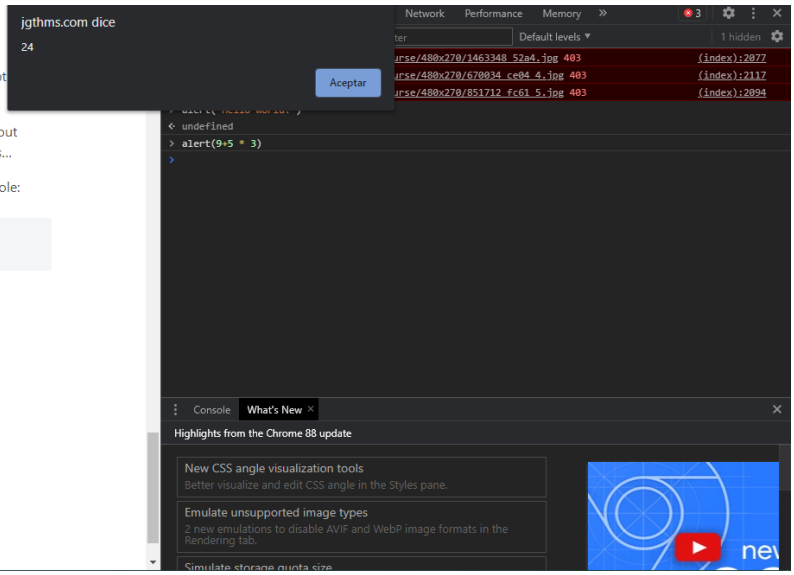
Like any other programming language, JavaScript handle **numbers**.

These numbers can be big or small, with or without decimals, combined through numeric operations...

Type or paste the following snippet in your console:

```
alert(9+5 * 3)
```

Copy to clipboard



Browser dimensions info

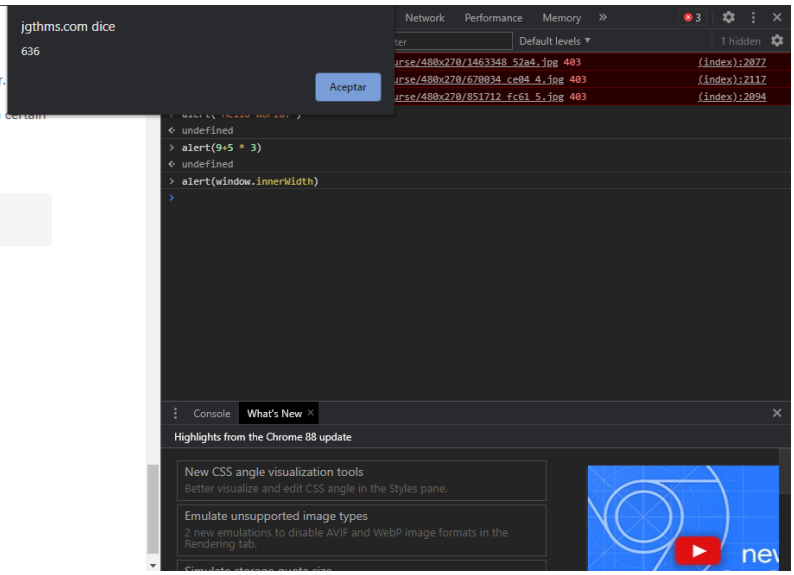
Numbers are everywhere! Especially in a browser.

For example, your current browser window has a certain **width** that you can access with JavaScript.

Type the following:

```
alert(window.innerWidth)
```

Copy to clipboard



Objects type

`window` is a JavaScript **object**.

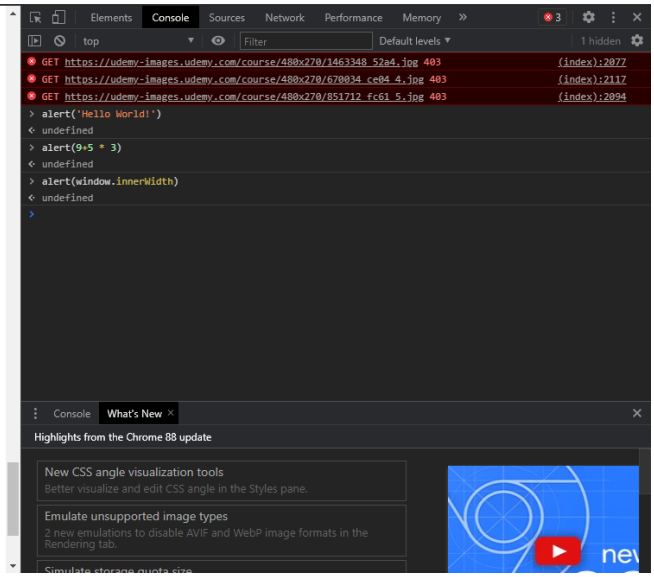
`innerWidth` is a **property** of the `window` object. To access this property, you used a dot `.` to specify you wanted the `innerWidth` that exists *within* the `window` object.

The JavaScript object is basically a **type** that contains other things.

For example, `window` also has:

- `window.origin` which is a string
- `window.scrollY` which is a number
- `window.location` which is an object

If `window` is an object that contains `location` which is another object, this means that JavaScript objects support...



Nesting info

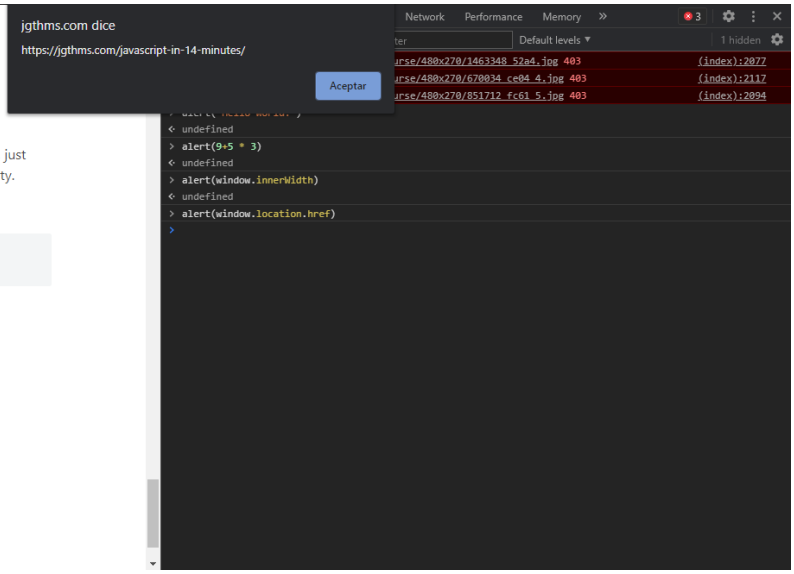
A property of an object can be an object itself (inception!).

Since `window.location` is an object too, it has properties of its own. To access these properties, just add another dot `.` and the name of the property.

For example, the `href` property exists:

```
alert(window.location.href)
```

Copy to clipboard



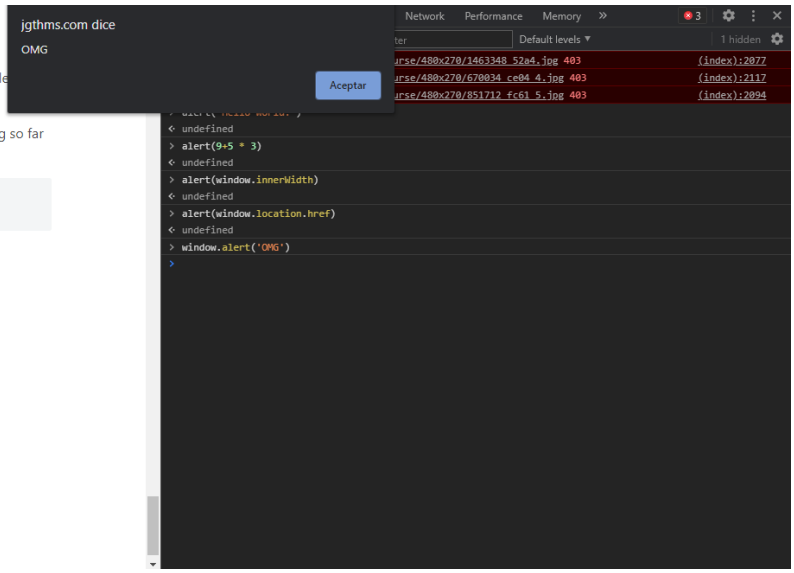
Methods info

When an object's **property** is a **function**, it's calle
method instead.

Actually, the `alert()` function we've been using so far is a method of the `window` object!

```
window.alert('OMG')
```

Copy to clipboard



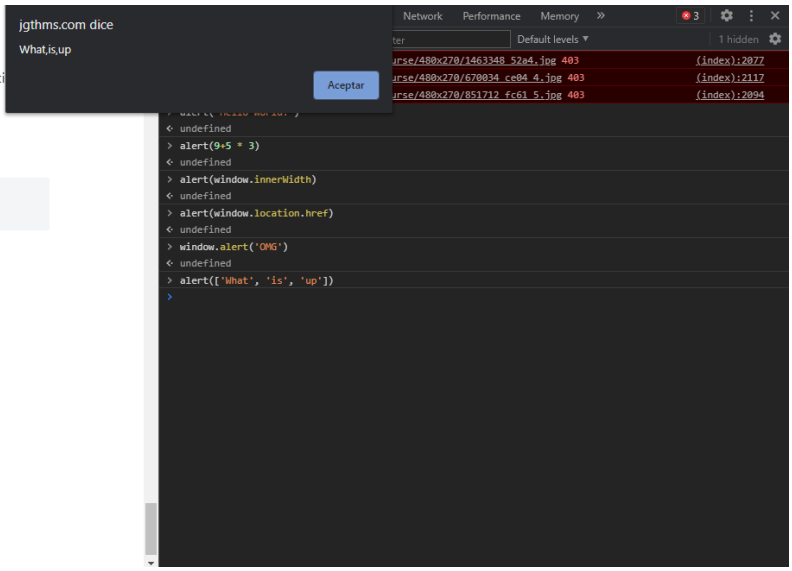
Arrays type

A JavaScript **array** is a type that can contain multi
values, as if they were in an ordered list.

Let's pass an array of **3 strings** to the `alert()` function:

```
alert(['What', 'is', 'up'])
```

Copy to clipboard

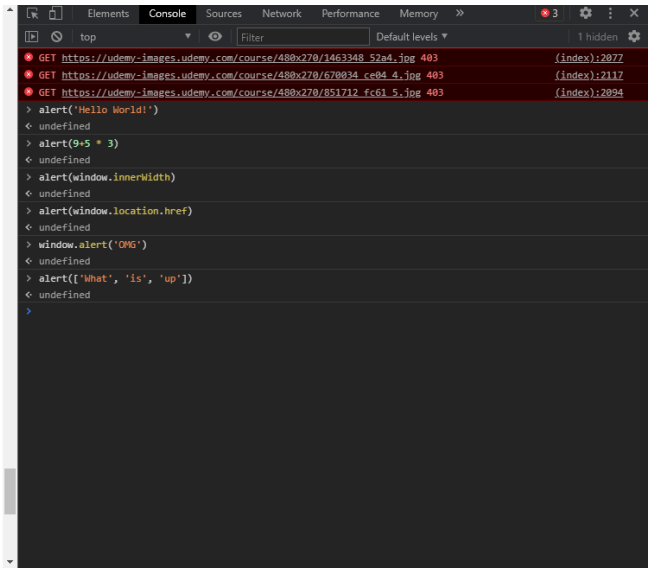


Exactly!

You already know the syntax for calling the `alert` function: `alert(argument)` .

In this case, the argument you passed is an array with 3 items that you have defined like this:

1	you opened a **square bracket**	`['What', 'is', 'up']`
2	you typed the **first item** of the array, a string	`['What', 'is', 'up']`
3	you typed a **comma** to separate the items	`['What', 'is', 'up']`
4	you added **two other items** to the list	`['What', 'is', 'up']`
5	you **closed** the square bracket	`['What', 'is', 'up']`



5	you **closed** the square bracket	`['What', 'is', 'up']`

An array can contain any **type** of values: strings, numbers, objects, other arrays, and *more*...

Try out this snippet:

```
alert([2 + 5, 'samurai', true])
```

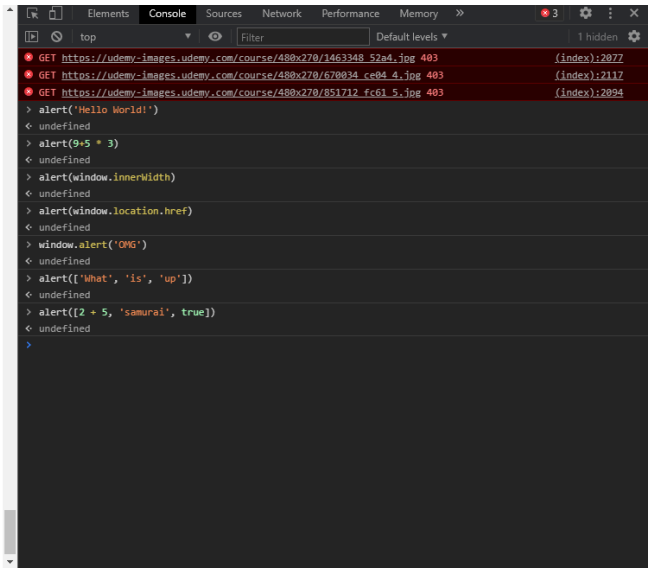
Copy to clipboard

True!

The first item `2 + 5` is a number, while the second one `'samurai'` is a string.

What about the **third argument**? It's not a string because it's not wrapped in quotes, and it's not a number either.

So what is it?

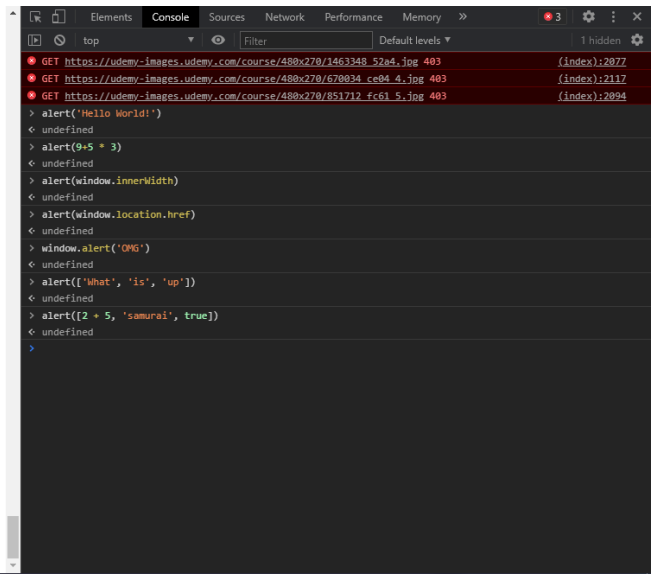


Booleans type

While strings and numbers have an infinite amount of *possible* values, a **boolean** can only be either `true` or `false`.

By combining the `alert()` function and the 3-item array on a *single* line, it makes our code less readable.

What if we could split the two by moving the array onto its own line?



Variables concept

We can move the array into a **variable**.

A variable is a **container** that stores a certain **value**. It has a **name** (so you can identify and re-use it), and it has a **value** (so you can update it later on).

```
var my_things = [2 + 5, 'samurai', true];
```

- `my_things` is the **name** of the variable
- `[2 + 5, 'samurai', true]` is the **value** of the variable

Here's the breakdown:

1 you typed the **keyword** `var`
```
var my_things = [2 + 5, 'samurai', true];
```

2 you typed the **name** of the variable
```
var my_things = [2 + 5, 'samurai', true];
```

3 you typed the **assignment** operator `=`
```
var my_things = [2 + 5, 'samurai', true];
```

4 you typed the **array**
```
var my_things = [2 + 5, 'samurai', true];
```

5 you typed a **semicolon** to end the statement
```
var my_things = [2 + 5, 'samurai', true];
```

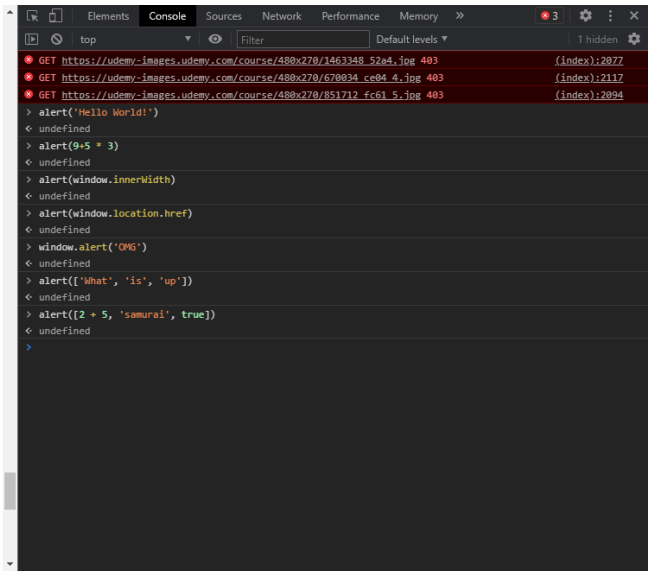
This means that `my_things` is equal to `[2 + 5, 'samurai', true]`.

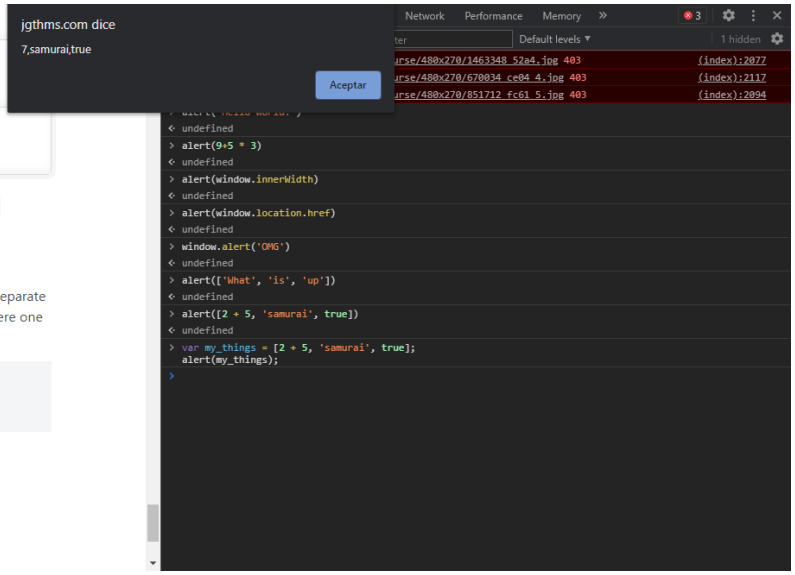
We can now reinstate the `alert` function:

Since we now have *two statements*, we need to separate them with a **semicolon** `;` in order to know where one ends and the next one begins.

```
var my_things = [2 + 5, 'samurai', true];
alert(my_things);
```

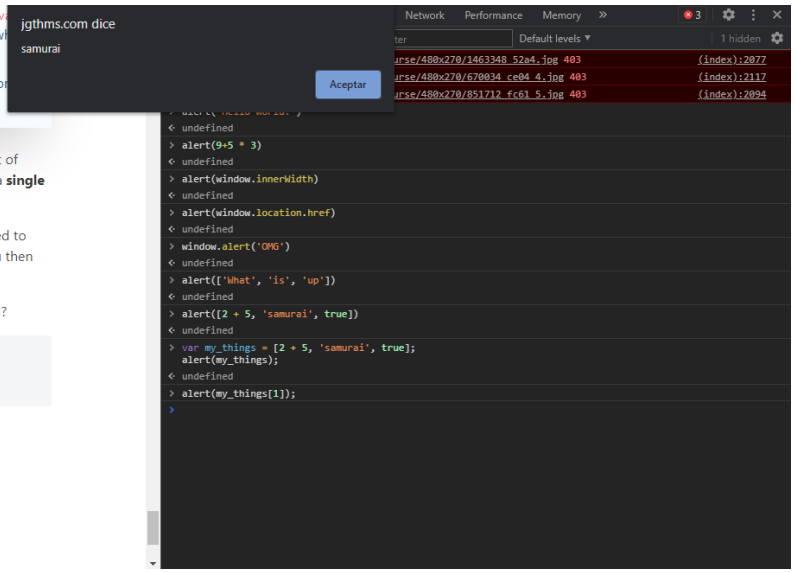
Copy to clipboard





So that's why I decided to use the keyword v
here since it's easier to use and understand wh
learning JavaScript.

If what I just said didn't make sense, don't wo
about it and keep on reading :-)



Although this `my_things` variable contains a list of several things, we might want to deal with only a **single** item of the list.

To access a *specific item* of an array, you first need to know its **index** (or position) within the array. You then need to wrap this index into **square brackets**.

Can you guess which item is gonna be displayed?

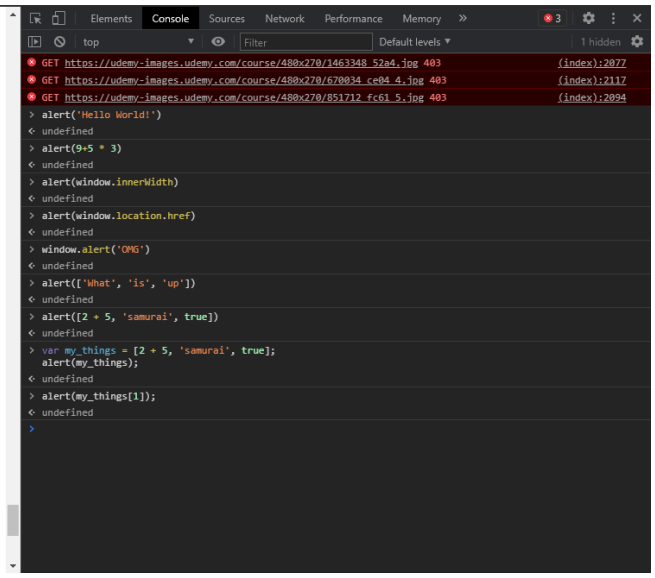
```
var my_things = [2 + 5, 'samurai', true];
alert(my_things[1]);
```

Copy to clipboard

You guessed it!

It's the **second item** that showed up! In programming, indexes start at zero `0`.

1. you typed the **name** of the array
   `my_things[1]`
2. you opened a **square bracket**
   `my_things[1]`
3. you typed the **index** of the item you wanted to access
   `my_things[1]`
4. you **closed** the square bracket
   `my_things[1]`



It turns out that variables are **objects** too! Which means that variables also have **properties** and **methods**.

For example, `my_things` has a property called `length`:

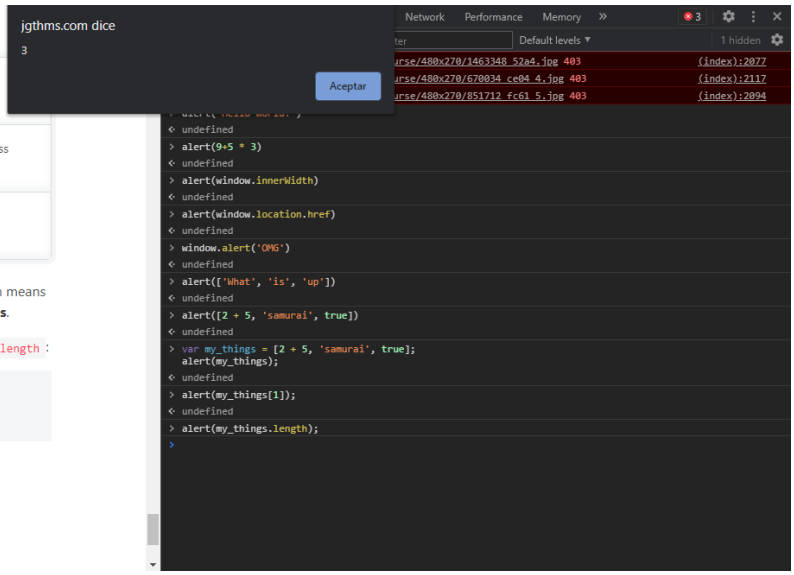
1. you typed the **name** of the array
   `my_things[1]`
2. you opened a **square bracket**
   `my_things[1]`
3. you typed the **index** of the item you wanted to access
   `my_things[1]`
4. you **closed** the square bracket
   `my_things[1]`

It turns out that variables are **objects** too! Which means that variables also have **properties** and **methods**.

For example, `my_things` has a property called `length`:

```
var my_things = [2 + 5, 'samurai', true];
alert(my_things.length);
```

Copy to clipboard



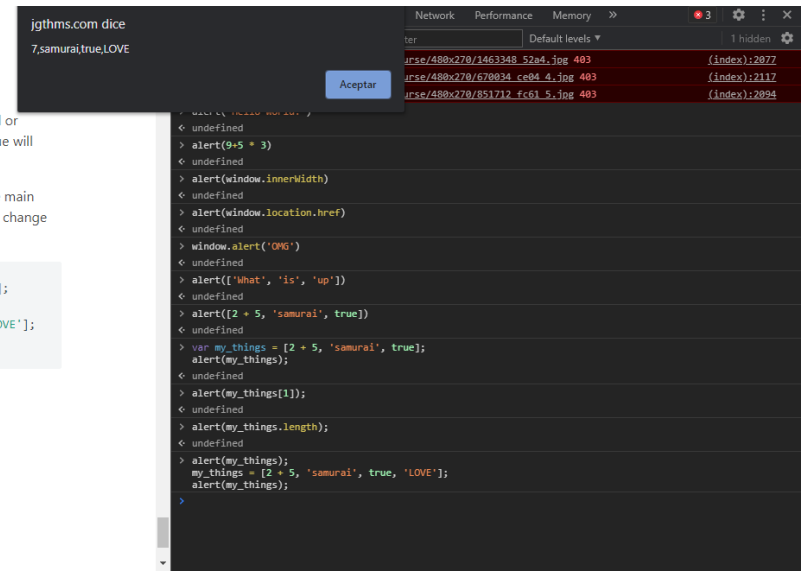
That's right!

The array has **3** items. You'll see that if you add or remove items in `my_things`, this `length` value will change accordingly.

While readability and properties are useful, the main point of a variable is that it's **editable**: you can change the value afterwards!

```
var my_things = [2 + 5, 'samurai', true];
alert(my_things);
my_things = [2 + 5, 'samurai', true, 'LOVE'];
alert(my_things);
```

Copy to clipboard



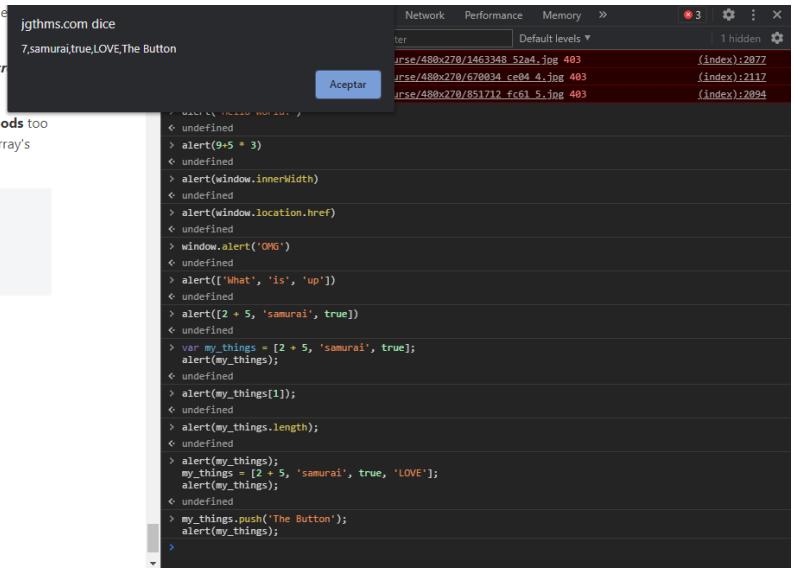
because we're editing the `my_things` variable de
two lines above.

**You only use the keyword `var` when you're *cr*
a new variable, not when you're *editing* one.

Do you remember I told you variables had **methods** too (since they're objects)? Another way to edit an array's value is by using the `push()` method:

```
var my_things = [2 + 5, 'samurai', true];
alert(my_things);
my_things.push('The Button');
alert(my_things);
```

Copy to clipboard



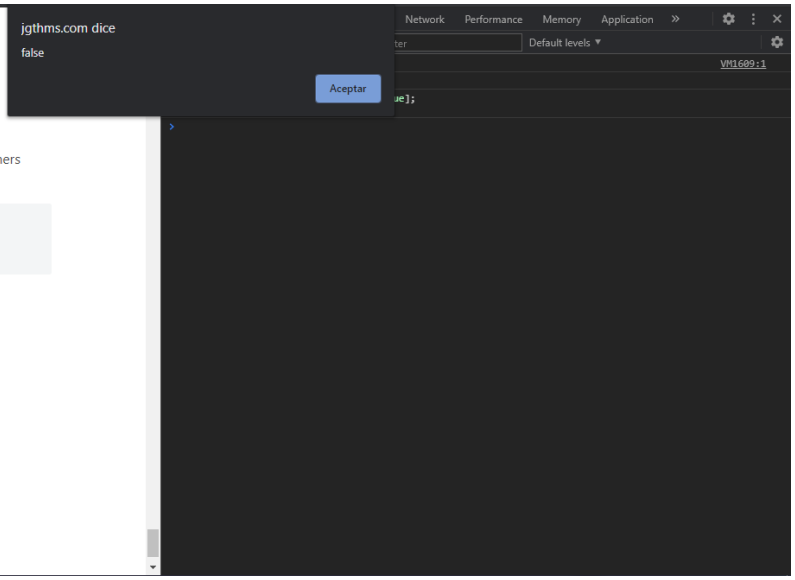
Fantastic!

The `my_things` array ends up with **4** items.

While the `push()` method altered the array, others simply **return** a value:

```
var my_things = [2 + 5, 'samurai', true];
alert(my_things.includes('ninja'));
```

Copy to clipboard



No ninja here!

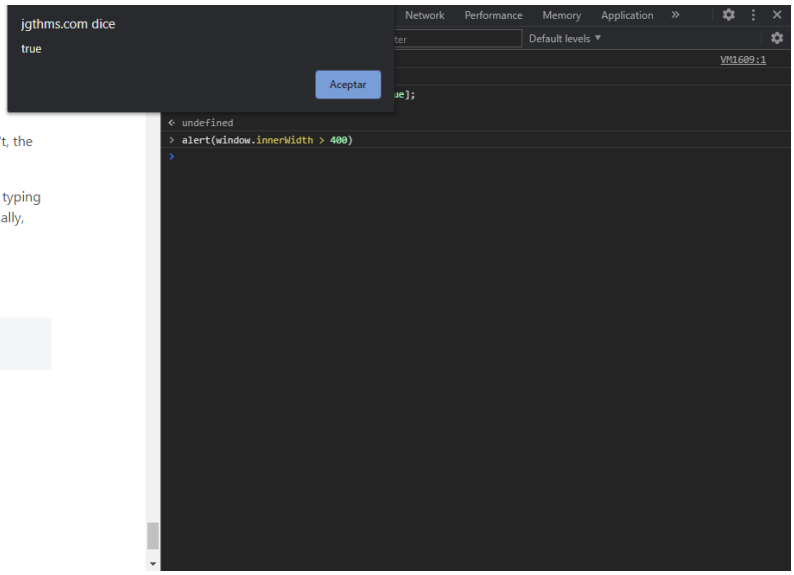
The `includes()` method checked if the string `'ninja'` was present in the array. Since it wasn't, the method returned `false`, a **boolean** value.

As a matter of fact, when dealing with booleans, typing the keywords `true` or `false` is quite rare. Usually, booleans exist as a result of a function call (like `includes()`) or a **comparison**.

Here's a "*greater than*" comparison:

```
alert(window.innerWidth > 400)
```

Copy to clipboard



Great again!

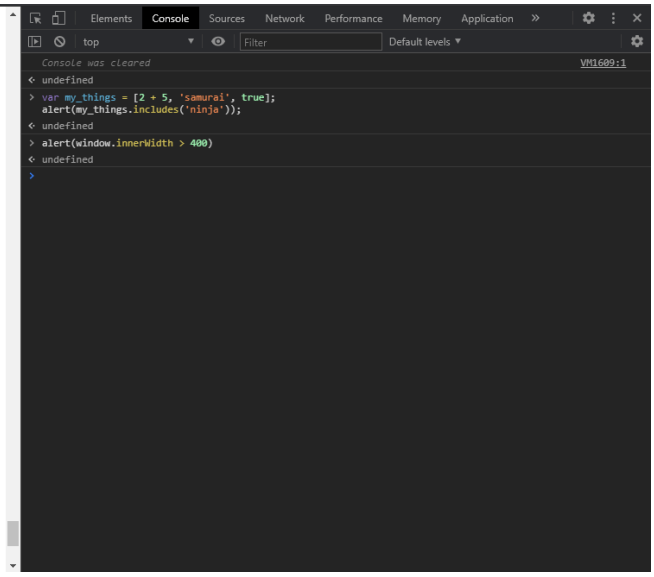
This means your browser window is wider than **400 pixels**.

1	you typed the **first item** to compare, a **number**	`window.innerWidth > 400`
2	you typed the "greater than" **operator**	`window.innerWidth > 400`
3	you typed the **second item**, also a **number**	`window.innerWidth > 400`

Just like the `+` and `*` before, `>` is a JavaScript **operator**.

When you make such a comparison with the "*greater than*" operator `>`, you obtain a **boolean** value.

Since the result of a **comparison** only has **2** outcomes (`true` or `false`), it's useful in cases where your code has to make a decision...



Conditional statements concep

Conditional statements are one of the most impo
concepts in programming. They allow your code t
perform certain commands *only if* certain conditio
met. These conditions can for example be based on:

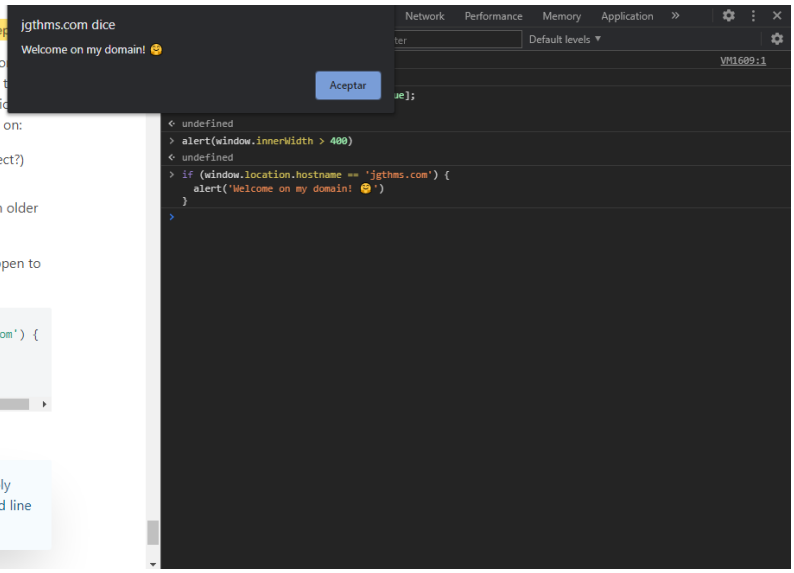
- a user's input (is the password entered correct?)
- the current state (is it day or night?)
- the value of a certain element (is this person older than 18?)

For now, let's trigger an alert box only if you happen to be on my domain!

```
if (window.location.hostname == 'jgthms.com') {
  alert('Welcome on my domain! 😊')
}
```

Copy to clipboard

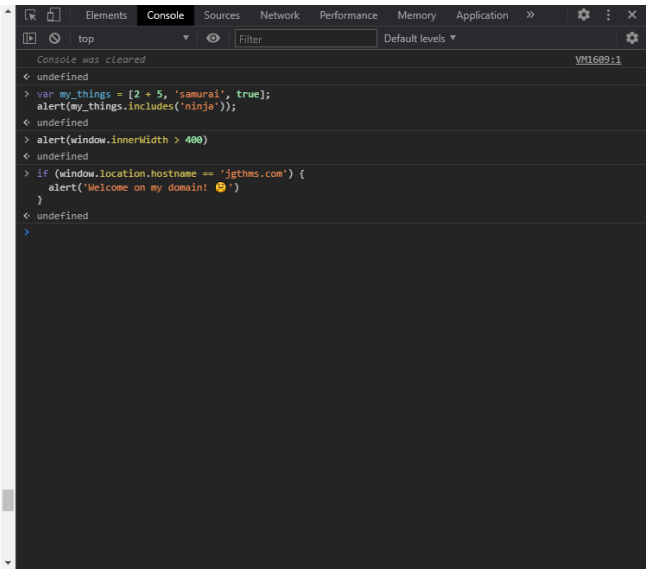
> If you want to **type** this code instead of simply copy-pasting it, press `Shift` + `Enter` to add line breaks in the console!



Equal indeed!

We're doing another comparison here, but with the "*equal to*" operator `==` instead.

1. you typed the **keyword** `if`
   ```
   if (window.location.hostname == 'jgthms.com')
   {
   ```
2. you opened a **parenthesis**
   ```
   if (window.location.hostname == 'jgthms.com')
   {
   ```
3. you typed a **comparison**
   ```
   if (window.location.hostname == 'jgthms.com')
   {
   ```
4. you **closed** the parenthesis
   ```
   if (window.location.hostname == 'jgthms.com')
   {
   ```
5. you opened a **curly bracket**
   ```
   if (window.location.hostname == 'jgthms.com')
   {
   ```



`'jgthms.com'`, we can use the "*not equal to*" ope
`!=`:

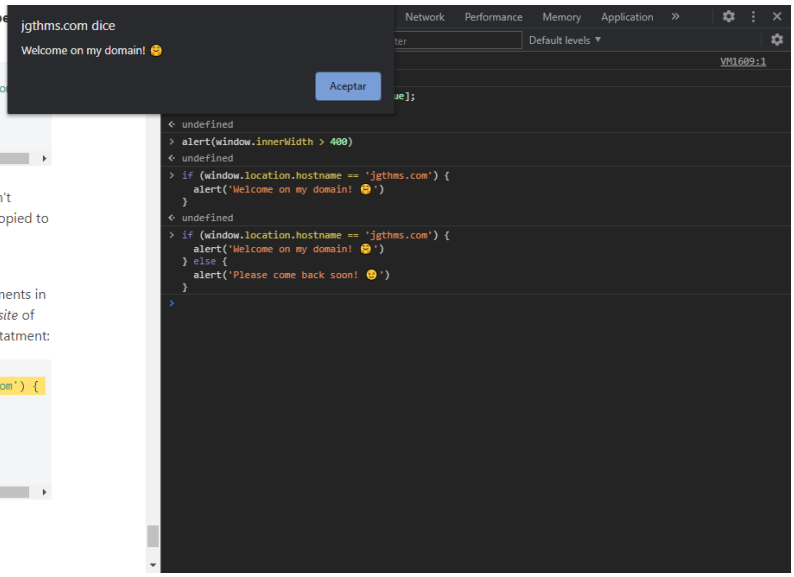
```
if (window.location.hostname != 'jgthms.co
  alert('Please come back soon! 😉')
}
```

If you try out this snippet, you'll see that it doesn't trigger anything! (Unless this tutorial has been copied to another domain 😀).

What if we wanted to handle both cases **simultaneously**? We could write two `if` statements in a row. But since one statement is the *exact opposite* of the other, we can **combine** them with a `else` statment:

```
if (window.location.hostname == 'jgthms.com') {
  alert('Welcome on my domain! 😊')
} else {
  alert('Please come back soon! 😉')
}
```

Copy to clipboard



Still equal!

With this conditional setup, we can be sure that:

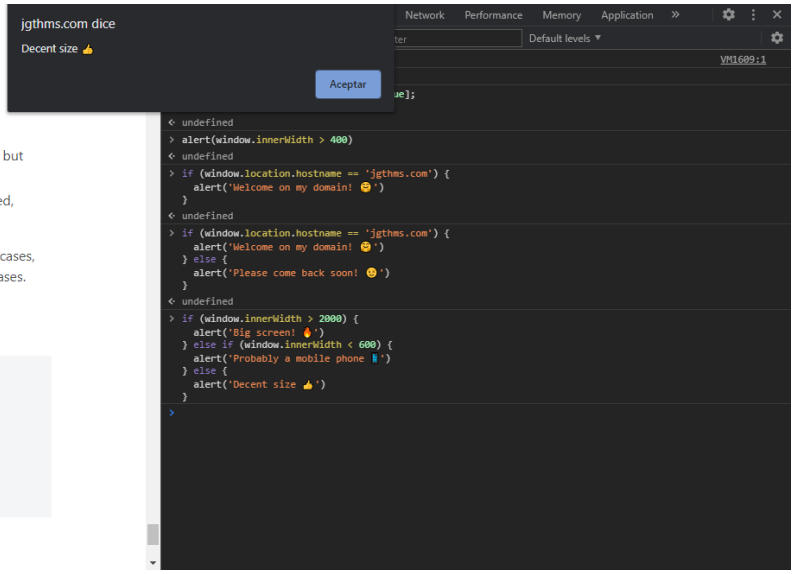
- **only one** of the two alerts will be triggered, but never both
- at least one of the two alerts *will* be triggered, because we're covering **all** cases

While `else` is useful for covering all *remaining* cases, sometimes you want to handle more than two cases.

By using `else if` you can add **intermediate** statements and handle multiple cases:

```
if (window.innerWidth > 2000) {
  alert('Big screen! 🔥')
} else if (window.innerWidth < 600) {
  alert('Probably a mobile phone 📱')
} else {
  alert('Decent size 👍')
}
```

Copy to clipboard



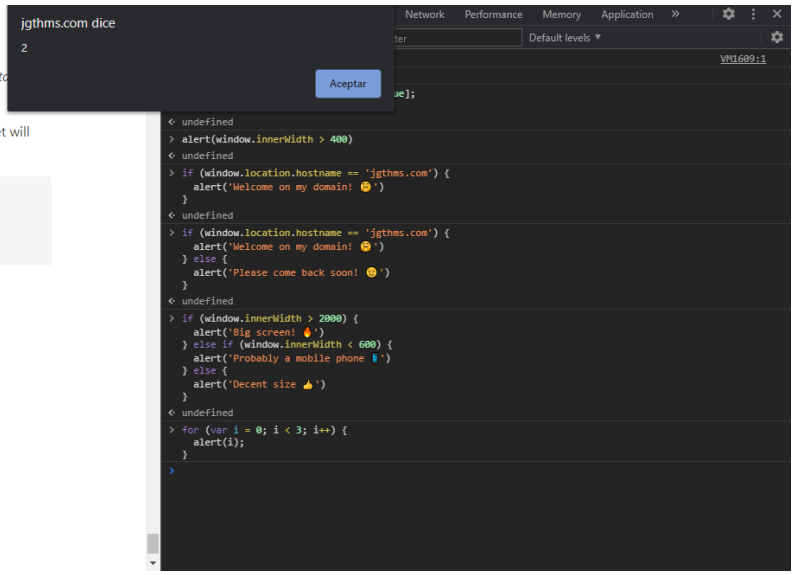
Loops concept

When you want to execute a block of code a *certa* *amount* of times, you can use a JavaScript **loop**.

Can you guess how many alert boxes this snippet will trigger?

```
for (var i = 0; i < 3; i++) {
  alert(i);
}
```

Copy to clipboard



Three it is!

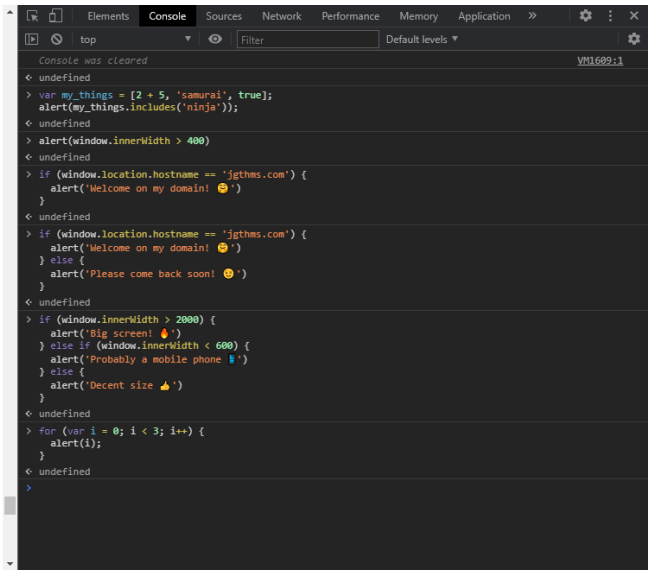
There were exactly **3** alert boxes! Let's dissect what happened:

- `var i = 0` is the **initial state**
  Before the loop even starts, the variable `i` is assigned a value of zero `0`.
- `i < 3` is the **conditional statement**
  On every iteration of the loop, we check this comparison.
  If it's `true`, we execute the code in the block.
  If it's `false`, we exit the loop.
- `i++` is the **increment expression**
  If the block of code is executed, this expression is executed.
  In this case, the value of `i` is incremented by 1.

Here's how you implemented it:

1 you typed the **keyword** `for`
`for (var i = 0; i < 3; i++) {`

2 you opened a **parenthesis**
`for (var i = 0; i < 3; i++) {`



Looping through arrays tool

Arrays are a perfect candidate for loops, because in programming we often want to repeat the **same action** for *each* item of an array.

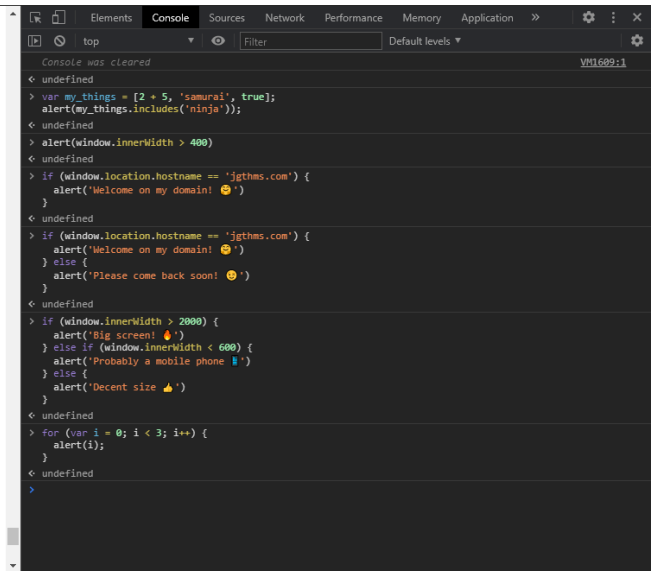
Let's say we wanted to trigger an alert box for *each* item of an array. While we could write as many `alert()` statements as there are items in the array (😰), this solution is cumbersome and ineffective! It would be prone to errors, and wouldn't scale at all with bigger arrays.

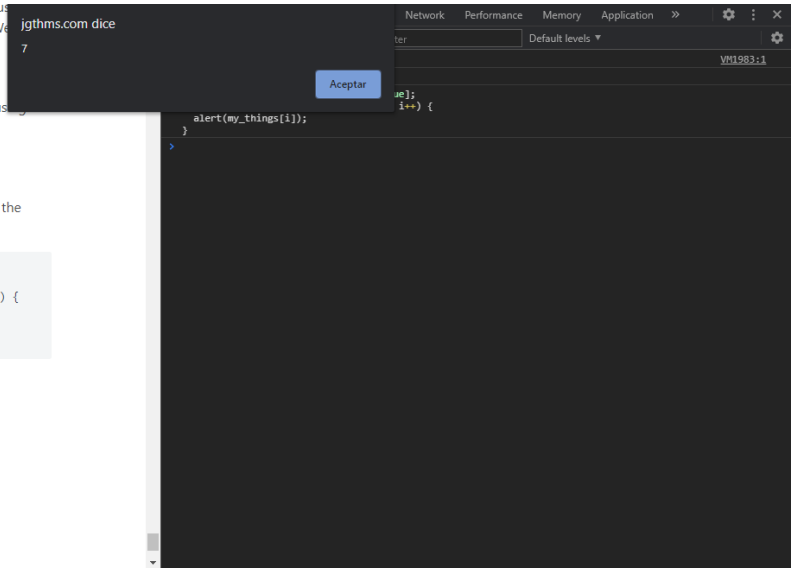
Since programming languages are here to help us simplify our work, let's figure out a better way. We already know a few things:

- We know how to get an array's **length**
- We know how to access an array's item by using the **index**
- We have access to the variable `i`, which conveniently increments 1 by 1

By combining these informations, we can devise the following snippet:

```
var my_things = [2 + 5, 'samurai', true];
```





Since programming languages are here to help us simplify our work, let's figure out a better way. We already know a few things:

- We know how to get an array's **length**
- We know how to access an array's item by using the **index**
- We have access to the variable `i`, which conveniently increments 1 by 1

By combining these informations, we can devise the following snippet:

```
var my_things = [2 + 5, 'samurai', true];
for (var i = 0; i < my_things.length; i++) {
  alert(my_things[i]);
}
```

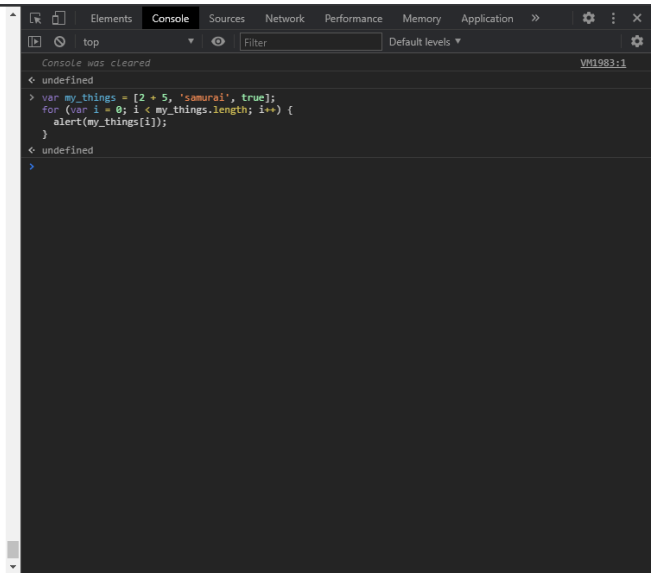
Copy to clipboard

In a row!

One by one, the items of the array were displayed in their own alert box.

While using a loop simplifies the process of going through each item of an array, we still had to create a `for` block **manually** and create a new variable `i` whose only purpose was to increment after each loop.

I know what you're thinking. "*There must be a better way!*"

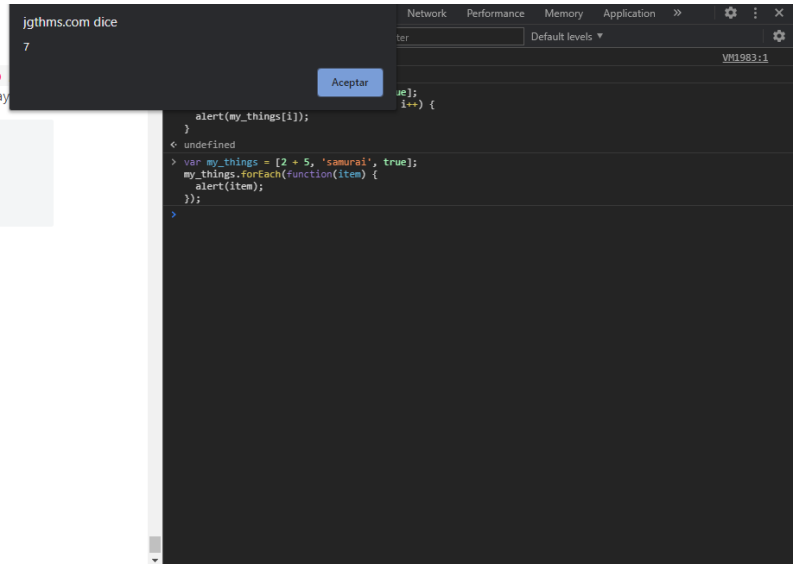


forEach loop tool

Arrays actually have a **method** called `forEach()` allows to perform a task *for each* item in the array

```
var my_things = [2 + 5, 'samurai', true];
my_things.forEach(function(item) {
  alert(item);
});
```

Copy to clipboard



Better!

Note a few improvements:

- There is no `i` variable involved
- We don't need to access the array's `length`
- We don't need to use the **index** with `my_thing[i]` to access the item

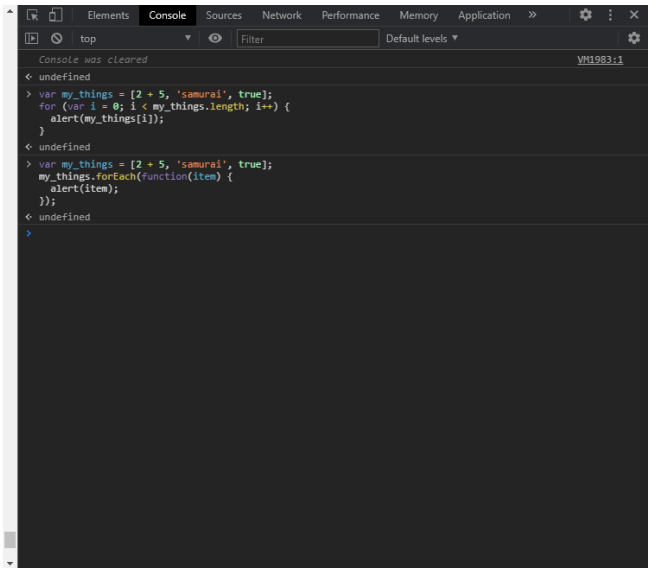
Remember the syntax of the `alert()` function? It was `alert(argument)`.

If you look carefully, you can see that `forEach()` has the exact same syntax! It's `forEach(argument)` but where the argument happens to be a **function** that spans 3 lines.

So far, we've used a few functions and methods:

- the `alert()` function (or window method)
- the `push()` array method
- the `includes()` array method
- the `forEach()` array method

We know how to **call** a function, but how do you actually create one?



Creating a custom function

The power of programming languages is the abili
create your **own functions**, that fit *your* needs.

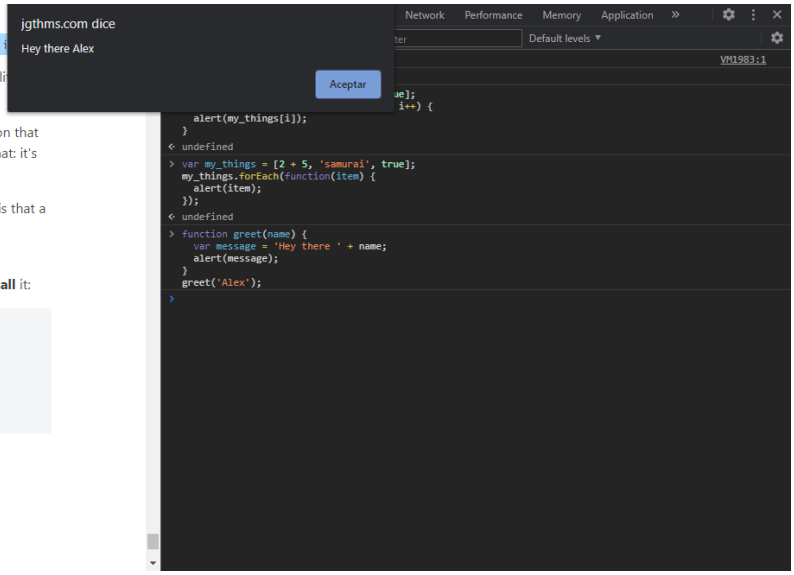
Remember the keyword/parentheses combination that we used for `if/else` and `for`? Well, guess what: it's *almost* the same pattern here!

I'm saying "almost" because the only difference is that a function needs a **name**!

Let's **create** a function called `greet()`, with **1** parameter called `name`, and then immediately **call** it:

```
function greet(name) {
  var message = 'Hey there ' + name;
  alert(message);
}
greet('Alex');
```

Copy to clipboard



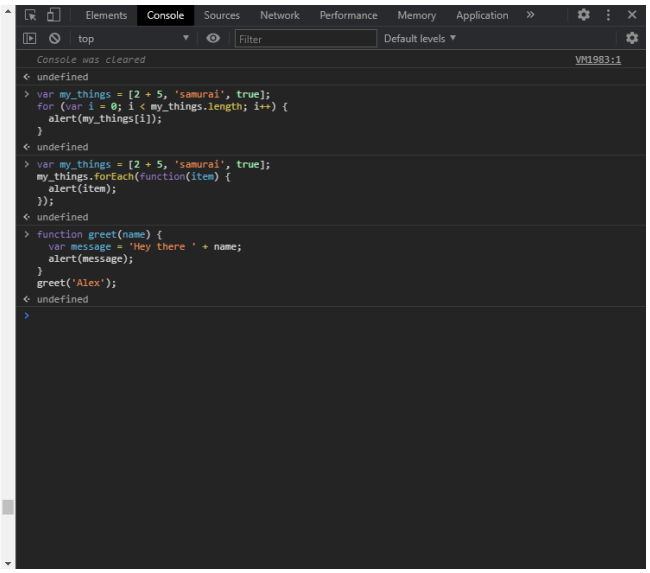
Greetings!

You've created your first function! It's a simple one but it can teach you a lot.

Note a few things:

- the **name** of the function is `greet`
- the **parameter** is called `name` : it's like a variable, since it acts as a container for a value
- we're creating a **variable** called `message` (a string) whose value is `'Hey there ' + name`
- what this **plus** sign `+` does is **concatenate** (or combine) the two strings to make a single longer one
- we're **calling** the `alert()` function, and use the `message` variable as parameter
- after having created the function, we're calling it with the argument `'Alex'`

> You might wonder why we're calling `name` a **parameter** when so far we've called **argument** the things we pass to a function. Well there's a difference!



Next steps victory!

Learn JavaScript

We've barely covered the basics here. But don't worry! There are *tons* of resources available online!

Here are a few **free** resources I'd recommend:

- Eloquent JavaScript
- freeCodeCamp
- The Modern JavaScript Tutorial
- Mozilla Developer Network
- You Don't Know JS

If you prefer **video tutorials**, check out these **Udemy courses** :

21.5 hours

Modern JavaScript From The Beginning

17.5 hours

The Complete JavaScript Course: Build a Real-World Project