Intro to Javascript

Functions

(Part 2)

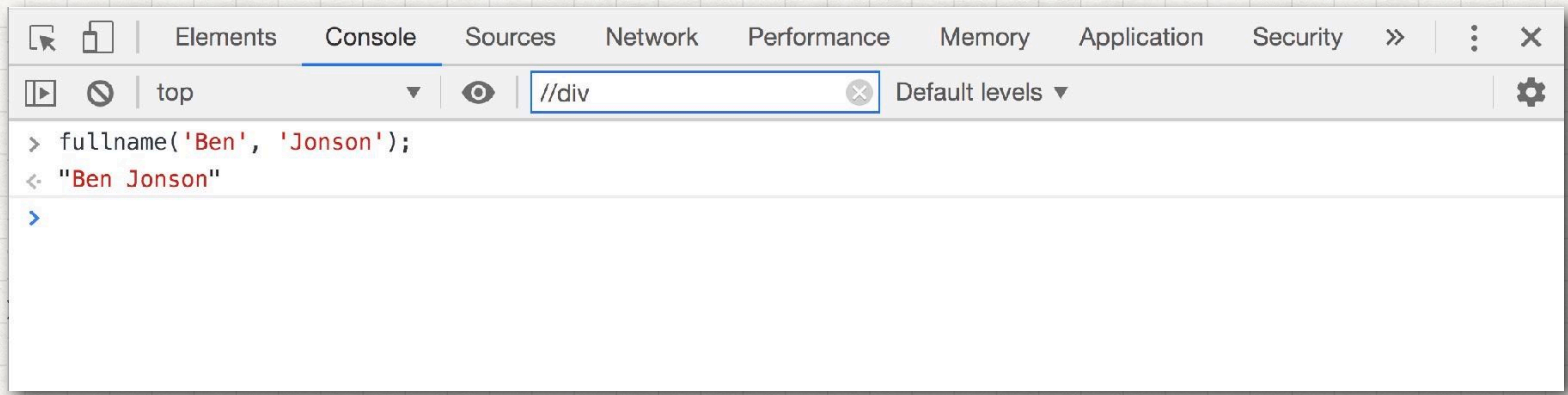
FUNCTIONS

We learned in the last slideshow that functions are blocks of code that have a name. In some senses, a function is like a variable—it just contains code rather than data. And once the browser “loads” a function, it’s always there for me to use:

```
<html>
<head><title>Functions!</title></head>

<body>
<script>
  function fullname(first, last) {
    var name = first + ' ' + last;
    return name;
  }
</script>
</body>
</html>
```

FUNCTIONS IN JAVASCRIPT

Once the browser loads that function, I can execute it from the console window:

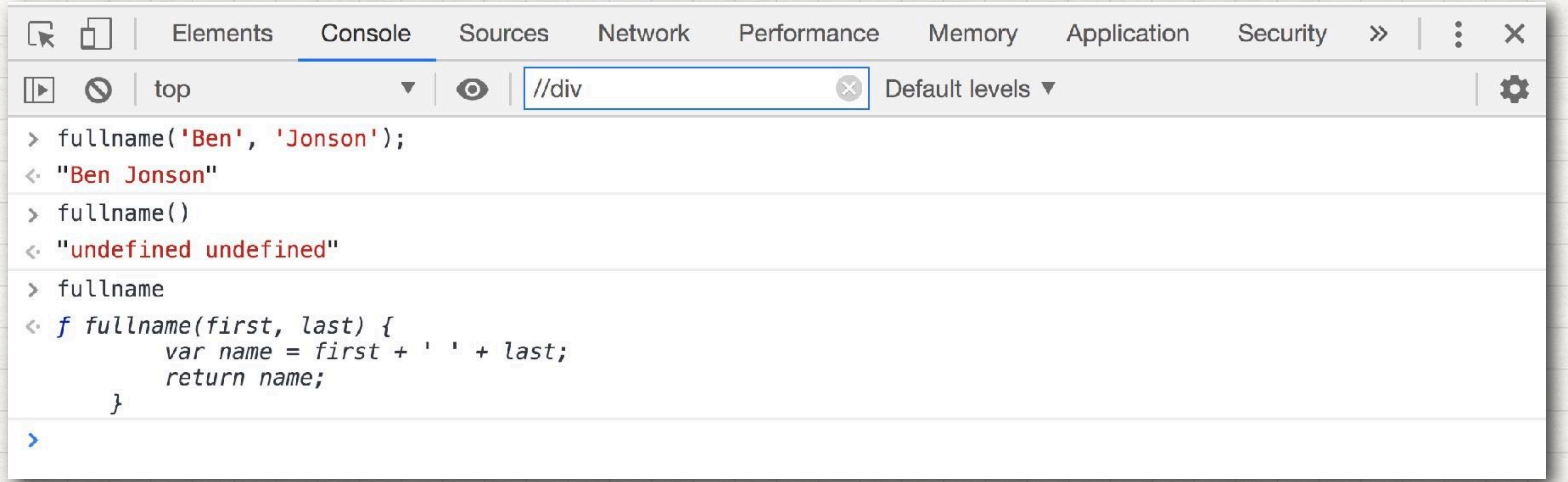


```
> fullname('Ben', 'Jonson');
<- "Ben Jonson"
>
```

FUNCTIONS IN JAVASCRIPT

Notice that if I execute the function with no input data, it still works but its returned value is undefined. The (and) parentheses act as a mini-verb, telling the browser to execute the function.

But if I "call" the function without the (and) parentheses, Javascript treats the name like a variable and shows me the contents—which, here, is the code:



```
> fullname('Ben', 'Jonson');
<· "Ben Jonson"
> fullname()
<· "undefined undefined"
> fullname
<· f fullname(first, last) {
          var name = first + ' ' + last;
          return name;
      }
>
```

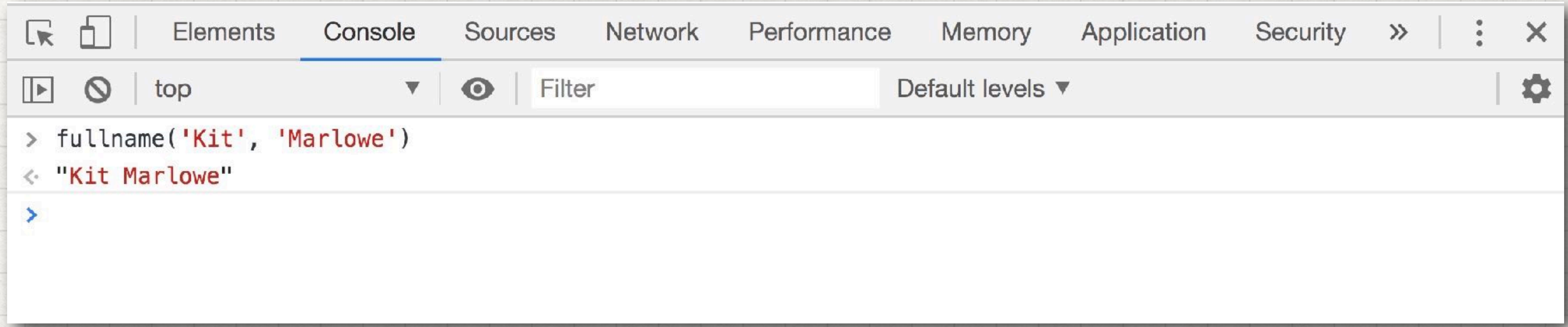
FUNCTIONS AS VARIABLES

I can write the same function in an alternate way by putting the code into a variable. It's really just a traditional variable assignment, nothing more:

```
<html>
<head><title>Functions!</title></head>

<body>
<script>
  var fullname = function(first, last) {
    var name = first + ' ' + last;
    return name;
  }
</script>
</body>
</html>
```

FUNCTIONS IN JAVASCRIPT

This new function works exactly the same way the other one did:



```
> fullname('Kit', 'Marlowe')
< "Kit Marlowe"
>
```

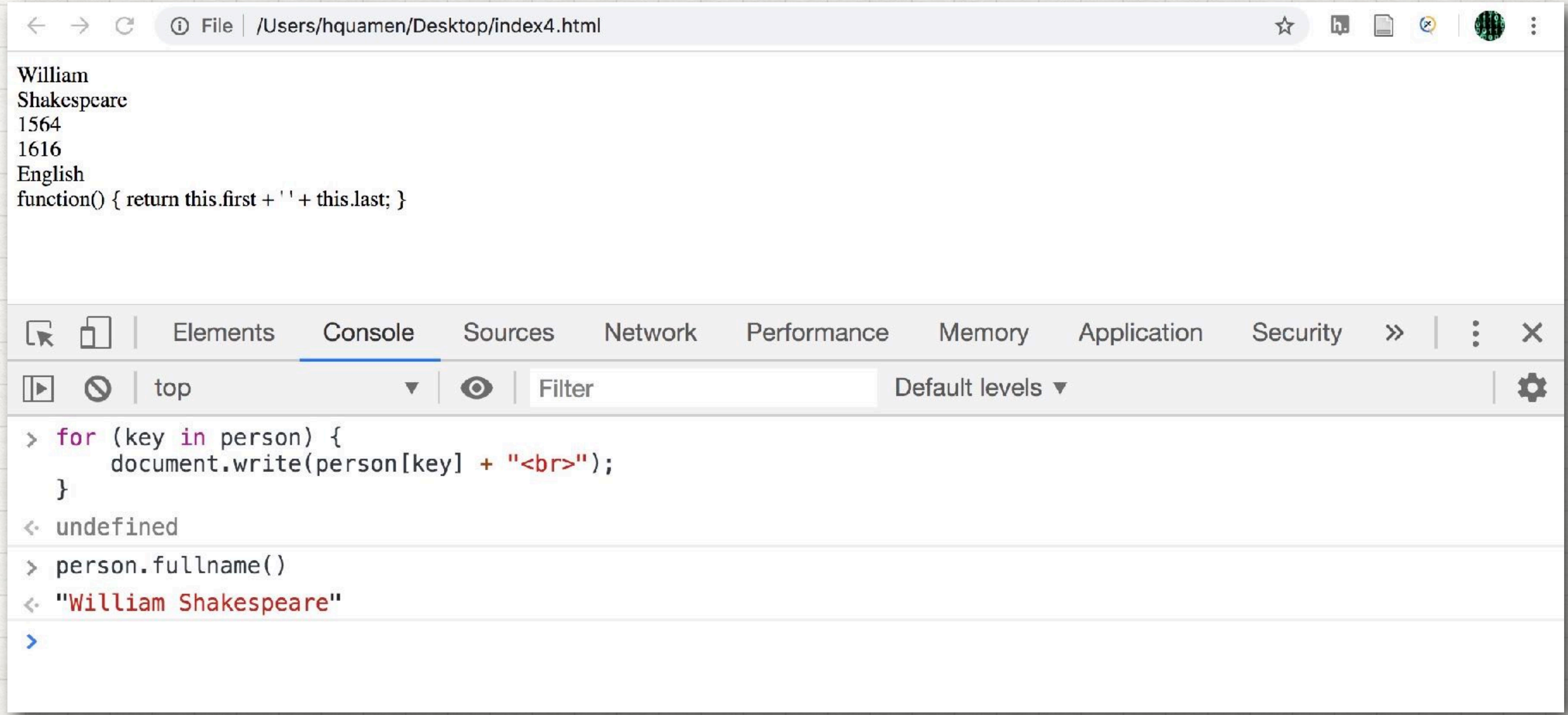
FUNCTIONS IN OBJECTS

That means we can actually put functions (or actions or behaviours) into objects:

```
<html>
<head><title>Functions!</title></head>

<body>
<script>
   var person = {
   'first': 'William',
   'last': 'Shakespeare',
   'birth': 1564,
   'death': 1616,
   'citizenship': 'English',
   'fullname': function() {
      return this.first + ' ' + this.last;
   }
}
</script>
</body>
</html>
```

FUNCTIONS IN OBJECTS

Now load that in the browser and play with it:

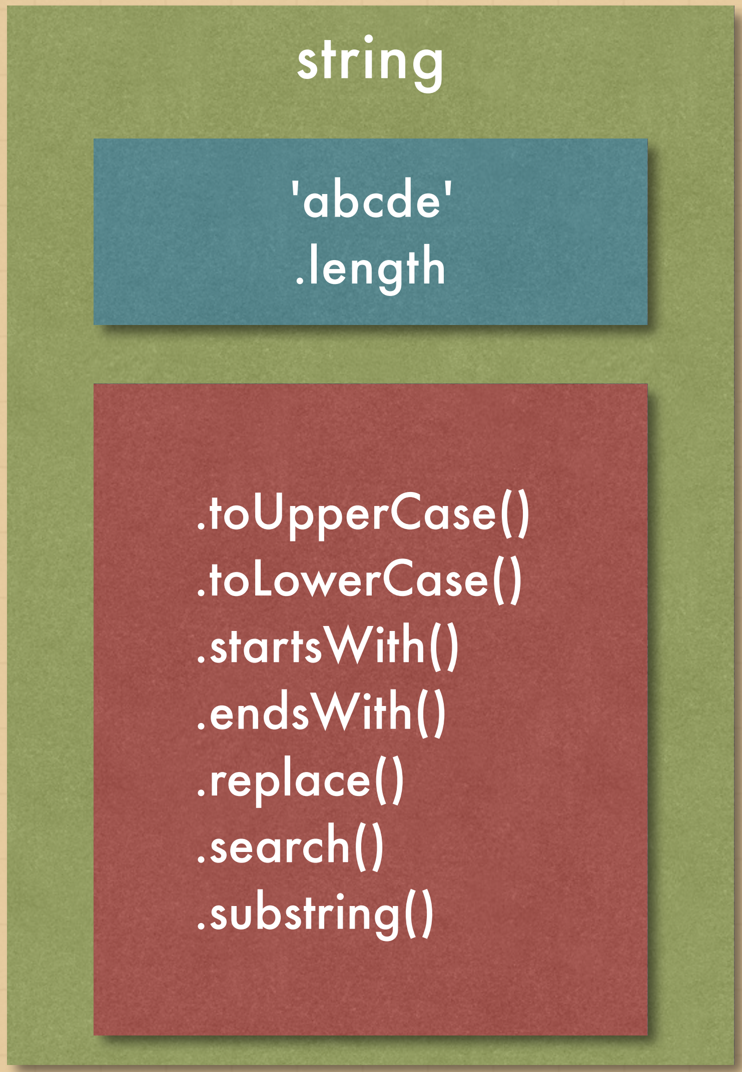


```
William
Shakespeare
1564
1616
English
function() { return this.first + ' ' + this.last; }
```

```
> for (key in person) {
      document.write(person[key] + "<br>");
  }
< undefined
> person.fullname()
< "William Shakespeare"
>
```

DIGRESSION: OBJECTS

Objects are a data structures comprised of two major parts: *properties* and *methods* — colloquially, *things the object knows* and *things the object does*. Both parts are bundled together, which makes objects very convenient and easy to use.

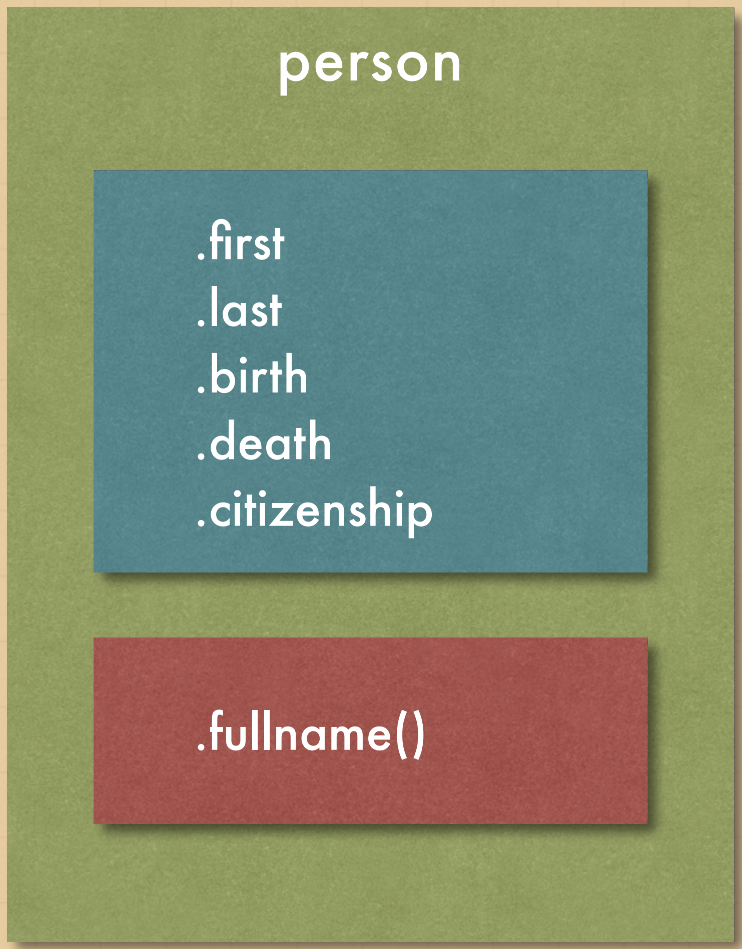


DATA: these are variables inside the object. Sometimes called properties or attributes, they are things the object knows.

ACTIONS or BEHAVIOURS: more typically called methods. They're things the object does. They're just functions, truthfully.

DIGRESSION: OBJECTS

We just built our own object, comprised of things it *knows* and things it *does*:



DATA: these are variables inside the object. Sometimes called properties or attributes, they are things the object knows.

ACTIONS or BEHAVIOURS: more typically called methods. They're things the object does. They're just functions, truthfully.

CALLBACK FUNCTIONS

Let's create a new Javascript library and put these functions in it:

```
// callback functions

function give_alert(message) {
   alert(message);
}

function console_log(message) {
   console.log(message)
}
```

Try this out in the browser and call these functions in the console window.

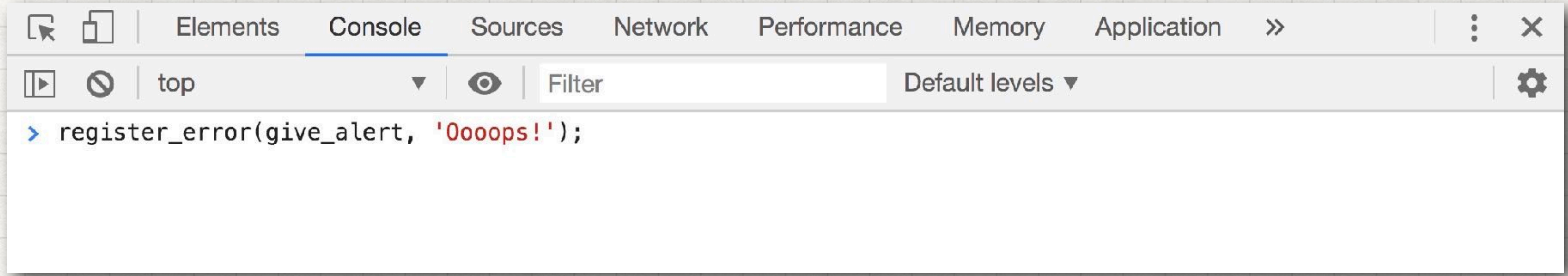
CALLBACK FUNCTIONS

Now add a new function that can handle any kind of error reporting technique:

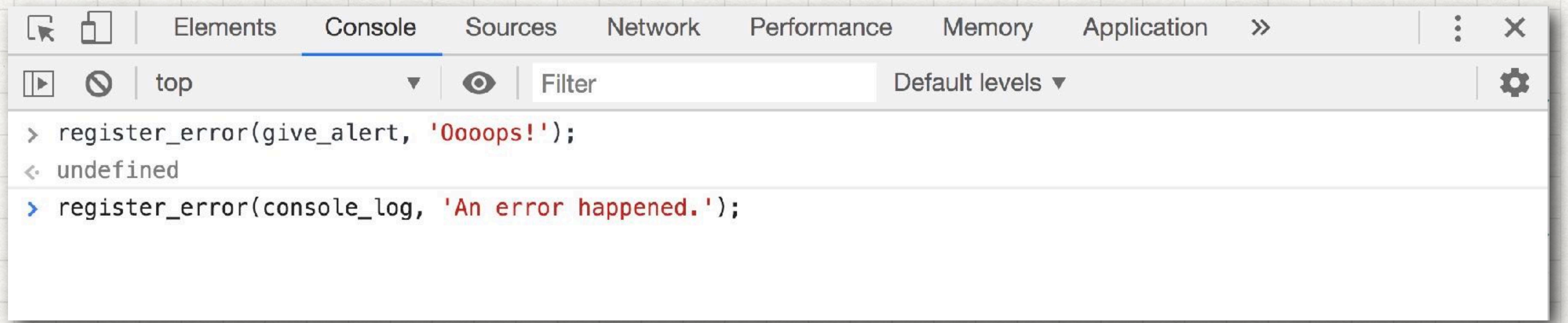
```
// callback functions

function give_alert(message) {
   alert(message);
}

function console_log(message) {
   console.log(message)
}

function register_error(method, message) {
   method(message);
}
```

CALLBACK FUNCTIONS

Now my "main program" can decide where to send errors. I can simulate this in the console window:



```
> register_error(give_alert, 'Oooops!');
```



```
> register_error(give_alert, 'Oooops!');
<· undefined
> register_error(console_log, 'An error happened.');
```

CALLBACK FUNCTIONS

Remember the two parameters that our function receives: the first is a function and the second is some data (probably a string):

```
function register_error(method, message) {
   method(message);
}
```

In other words:

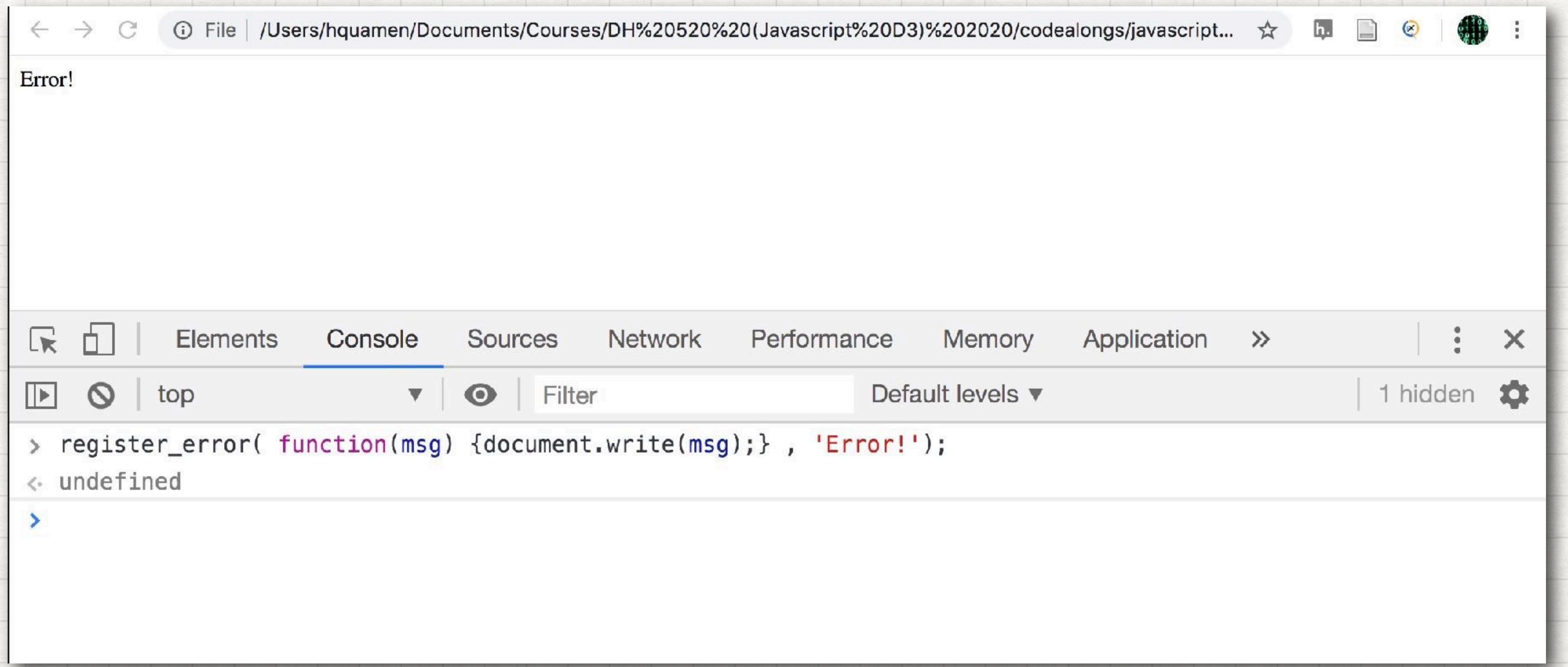
```
register_error( function , string );
```

But what if I wanted to register an error elsewhere? I could write a nameless function directly into the parameter list:

```
register_error( function(msg) {document.write(msg);} , 'Error!' );
```

ANONYMOUS FUNCTIONS

This is called an *anonymous function* and it's a technique that D3 uses all the time.


File
/Users/hquamen/Documents/Courses/DH%20520%20(Javascript%20D3)%202020/codealongs/javascript...
Elements
Console
Sources
Network
Performance
Memory
Application
top
Filter
Default levels
1 hidden

Error!

```
> register_error( function(msg) {document.write(msg);} , 'Error!');
<· undefined
>
```

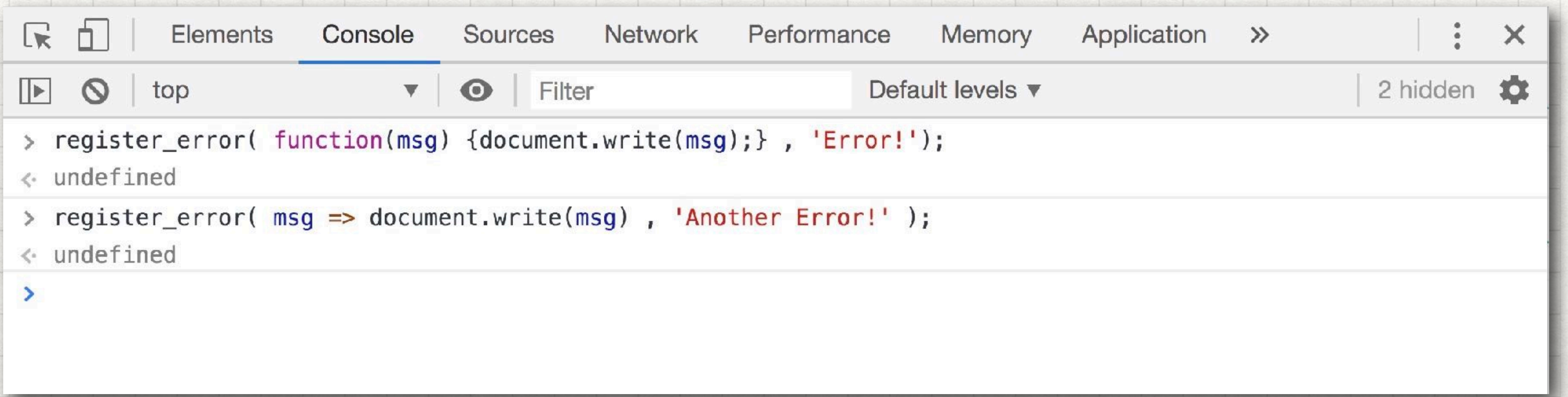
ARROW SYNTAX

In fact, anonymous functions are created so often in Javascript that the W3C has recently added a new syntax to make them even simpler. It's called arrow syntax:

```
register_error( function(msg) {document.write(msg);} , 'Error!' );
```

Can be written like this:

```
register_error( msg => document.write(msg) , 'Error!' );
```



```
> register_error( function(msg) {document.write(msg);} , 'Error!');
<· undefined
> register_error( msg => document.write(msg) , 'Another Error!' );
<· undefined
>
```

FOREACH

Arrays and objects have a **`forEach()`** method that iterates all their elements. This technique is slightly different because we need to pass a callback function to the method. In effect, we put the body of the loop into a function and send it to **`forEach():`**

```
letters = "ABCDEFGH".split("");
letters.forEach( item => console.log(item) );
```

Here, I’ve used the arrow syntax to write my function, but I could also write it in a more traditional format or I can put a function name in there.

RECAP: PASSING FUNCTIONS

So I can send functions around as if they were data. There are three ways I can do this. All are acceptable, so I can make the best choice at any given time:

1.) As a named function:

```
register_error(give_alert , 'Error!' );
```

2.) As an anonymous function written in the traditional way:

```
register_error( function(msg) {document.write(msg);} , 'Error!' );
```

3.) As an anonymous function written with the arrow syntax:

```
register_error( msg => document.write(msg) , 'Error!' );
```

D3 ARCHITECTURE

As it turns out, callback functions are game-changers. No longer is data the only thing that's moveable and portable. Now, *behaviours / actions / verbs* are also moveable. I can send an action to another object and can say, "When this thing happens, I want you to behave this way."

D3 uses this technique all the time. D3 says, in effect:

When a browser event happens to this DOM object, I want you to behave according to this function.

So to use D3 effectively, we need to know three things:

- DOM Tree Objects
- Browser Events
- Callback Functions

Intro to Javascript

Functions

(Part 2)