

# **Javascript**

**Because ECMAScript sounds horrible**

# Javascript

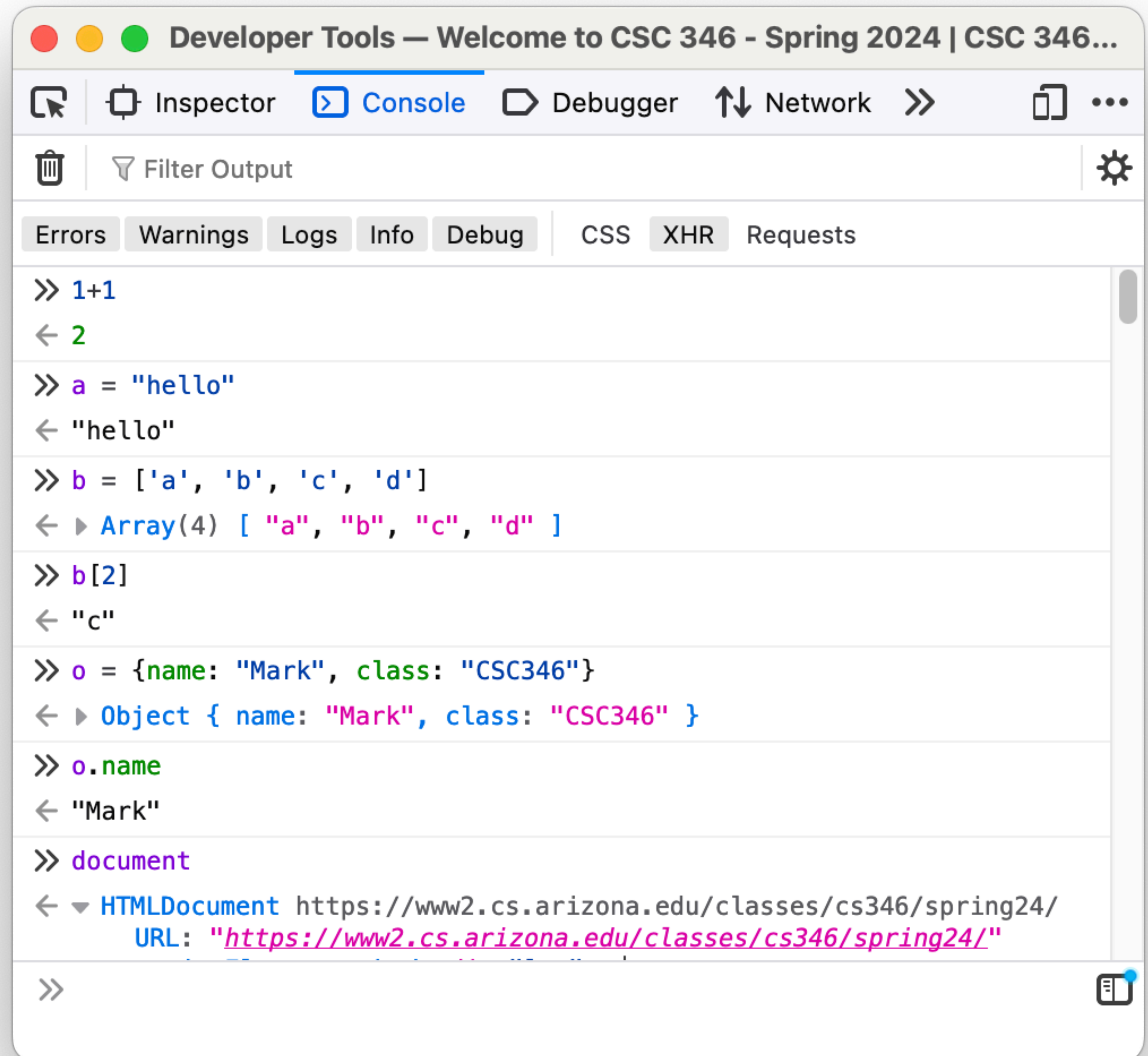
## Javascript is a general purpose programming language

- It usually runs within a browser
  - Node.js runs Javascript in a server / application context
- Developed in the mid nineties as a simple way to provide interactivity to web pages.
- Originally developed by Brendan Eich working at Netscape
- Submitted to ECMA standards body in 1996
- ECMAScript 5.1 released in 2011

# Javascript In A Browser

- REPL
  - Read-Eval-Print Loop
- All major browsers have a Javascript REPL system in the console

# Javascript In A Browser



The screenshot shows a browser window with the title "Developer Tools — Welcome to CSC 346 - Spring 2024 | CSC 346...". The "Console" tab is active, displaying a series of JavaScript commands and their corresponding outputs. The console is organized into sections: Errors, Warnings, Logs, Info, Debug, CSS, XHR, and Requests. The current view shows the following interactions:

```
>> 1+1
< 2

>> a = "hello"
< "hello"

>> b = ['a', 'b', 'c', 'd']
< ▶ Array(4) [ "a", "b", "c", "d" ]

>> b[2]
< "c"

>> o = {name: "Mark", class: "CSC346"}
< ▶ Object { name: "Mark", class: "CSC346" }

>> o.name
< "Mark"

>> document
< ▼ HTMLDocument https://www2.cs.arizona.edu/classes/cs346/spring24/
  URL: "https://www2.cs.arizona.edu/classes/cs346/spring24/"

>>
```

# Documentation

[http://ecma262-5.com/ELS5\\_HTML.htm](http://ecma262-5.com/ELS5_HTML.htm)

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>

# Data Types

## Basic Data Types

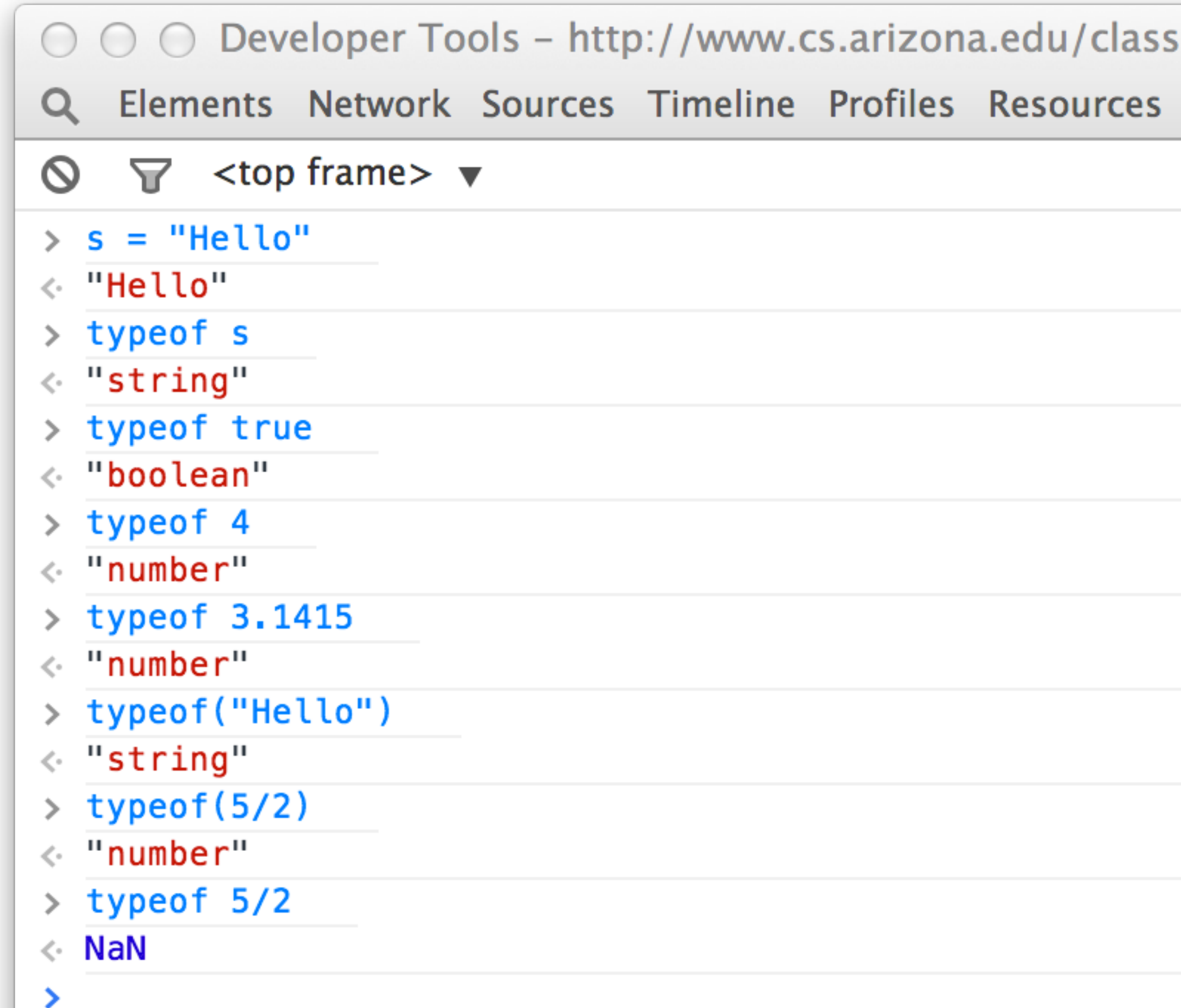
- number
- boolean
- string
- function
- object



# Data Types

## typeof

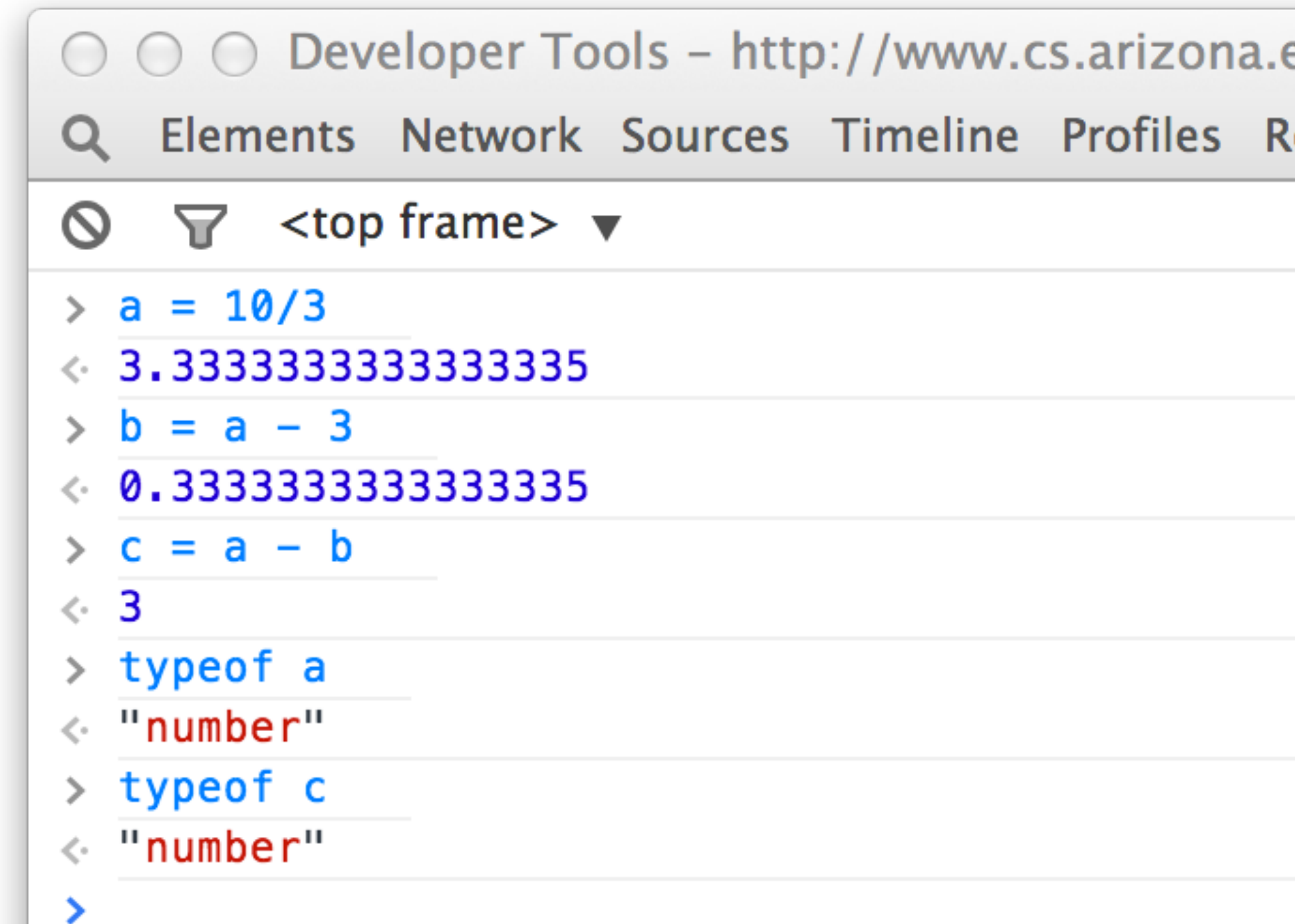
- `typeof` unary operator
- lets us know what we're dealing with
- If you're evaluating a complex operation, you need parenthesis. Not because `typeof` is a function, but to make sure that there's only one argument to `typeof`



```
Developer Tools - http://www.cs.arizona.edu/class
Elements Network Sources Timeline Profiles Resources
<top frame>
> s = "Hello"
< "Hello"
> typeof s
< "string"
> typeof true
< "boolean"
> typeof 4
< "number"
> typeof 3.1415
< "number"
> typeof("Hello")
< "string"
> typeof(5/2)
< "number"
> typeof 5/2
< NaN
>
```

# Numbers

- Javascript has a single number datatype to deal with all numbers.
- No distinction between integers, floats, doubles, etc.
- All numbers are represented as floating point numbers, but if the fractional part is zero, they're shown as integers.

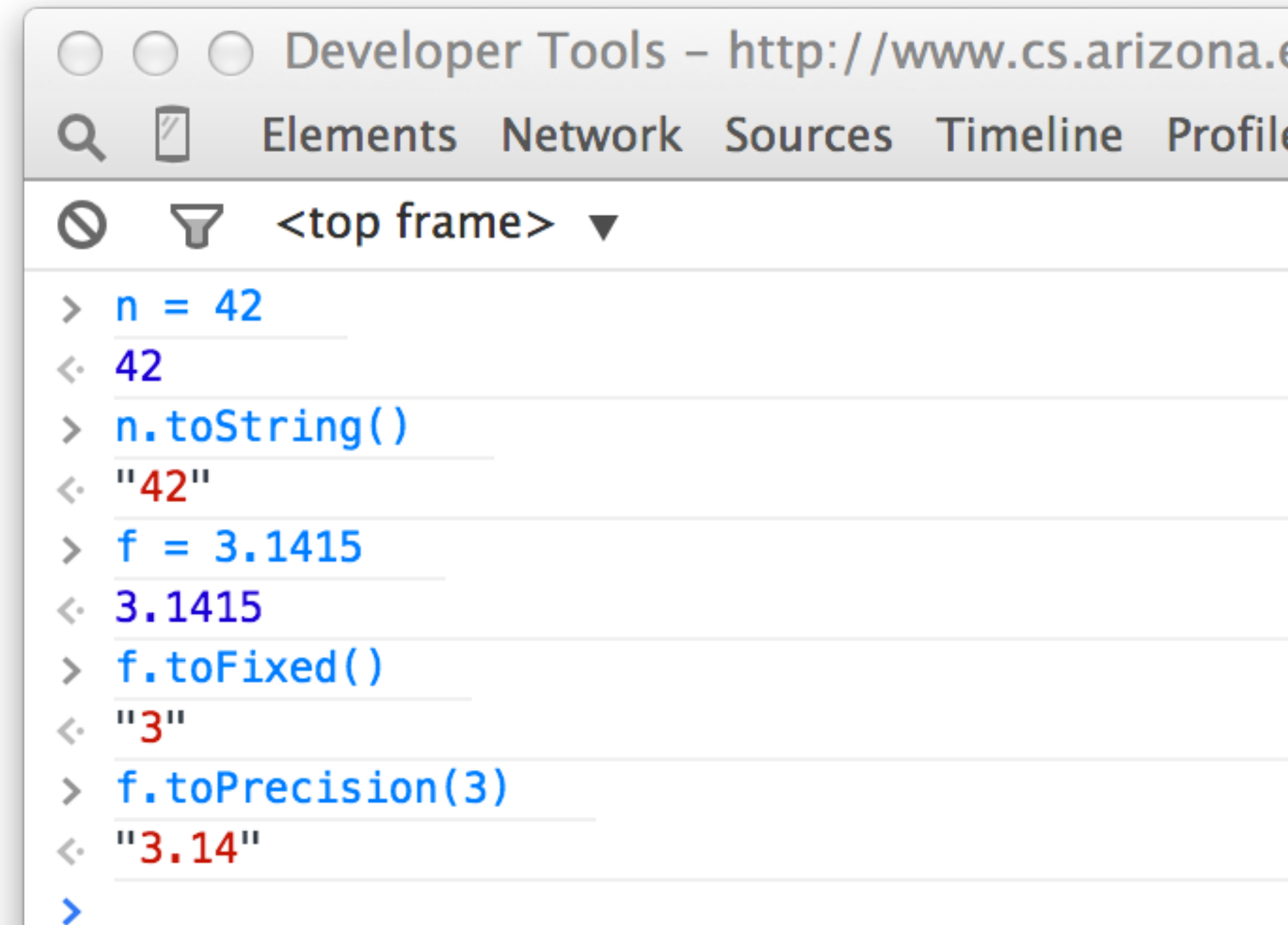


```
Developer Tools - http://www.cs.arizona.edu
Elements Network Sources Timeline Profiles R
<top frame>
> a = 10/3
< 3.3333333333333335
> b = a - 3
< 0.3333333333333335
> c = a - b
< 3
> typeof a
< "number"
> typeof c
< "number"
>
```



# Numbers

- Numbers stored in variables are converted objects when needed, to have methods and properties
- `Number.toString()`
- `Number.toPrecision()`

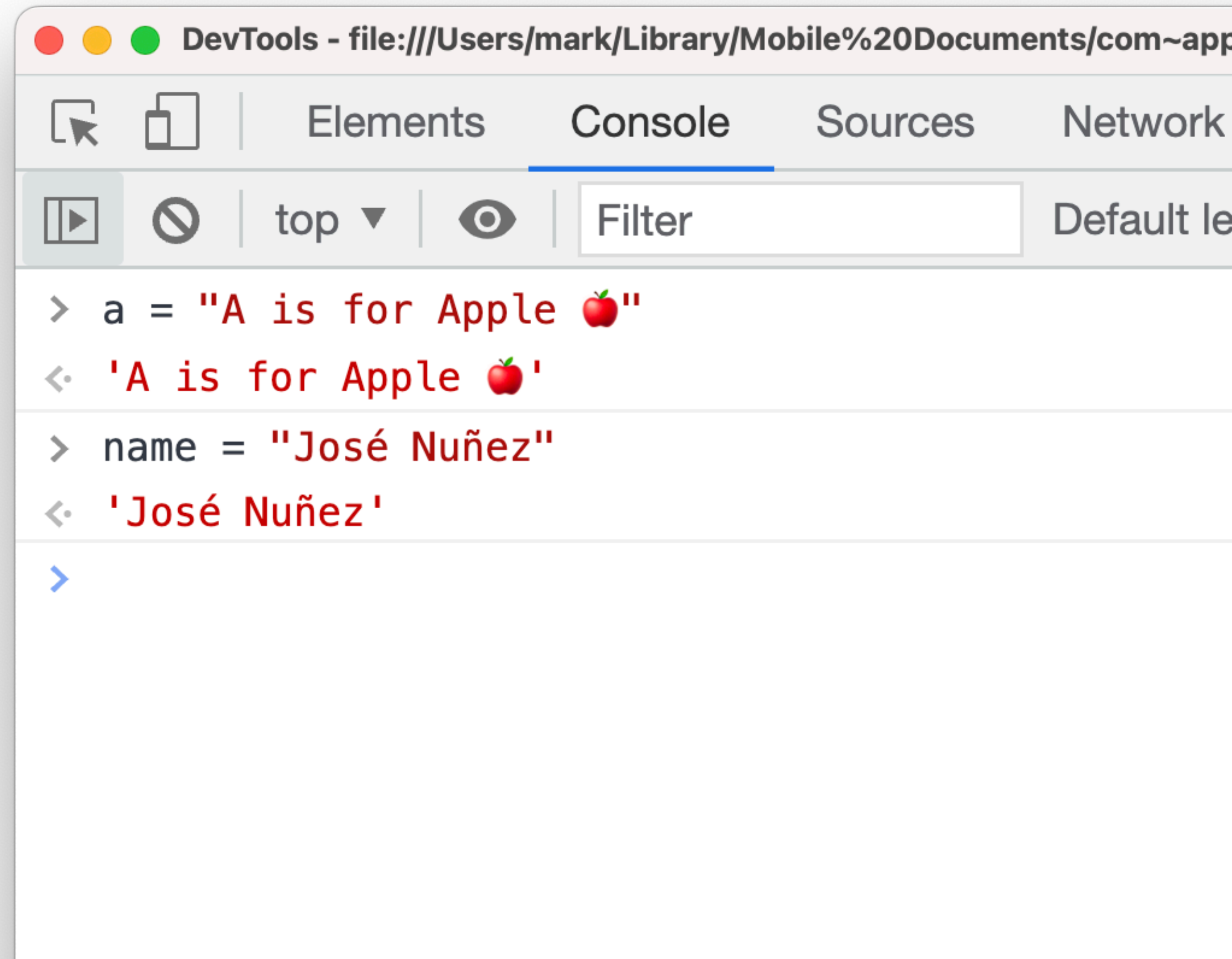


The screenshot shows a browser's developer tools console. The address bar displays 'Developer Tools - http://www.cs.arizona.edu'. The console has tabs for 'Elements', 'Network', 'Sources', 'Timeline', and 'Profile'. The current view is the console, showing a series of commands and their outputs:

```
> n = 42
< 42
> n.toString()
< "42"
> f = 3.1415
< 3.1415
> f.toFixed()
< "3"
> f.toPrecision(3)
< "3.14"
>
```

# Strings

- A series of zero or more characters.
- Unicode support is pretty good.

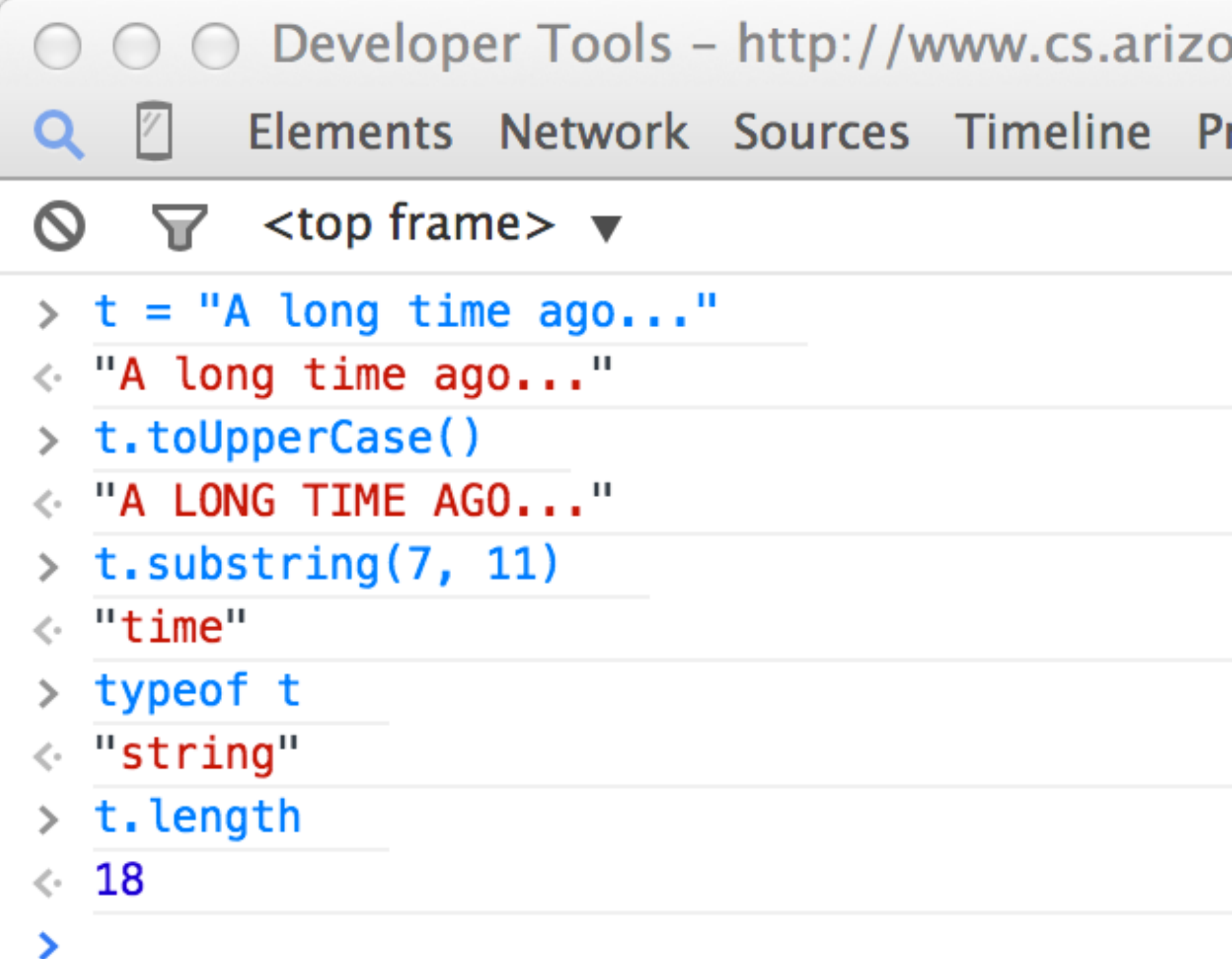


The screenshot shows the Chrome DevTools Console with the 'Console' tab selected. The console displays two JavaScript commands and their corresponding outputs. The first command is `a = "A is for Apple 🍏"`, which outputs `'A is for Apple 🍏'`. The second command is `name = "José Nuñez"`, which outputs `'José Nuñez'`. The console interface includes a toolbar with icons for back, forward, and search, a dropdown menu set to 'top', a search filter box, and a 'Default log' button. The window title is 'DevTools - file:///Users/mark/Library/Mobile%20Documents/com~app'.

```
> a = "A is for Apple 🍏"  
↵ 'A is for Apple 🍏'  
  
> name = "José Nuñez"  
↵ 'José Nuñez'  
  
>
```

# Strings

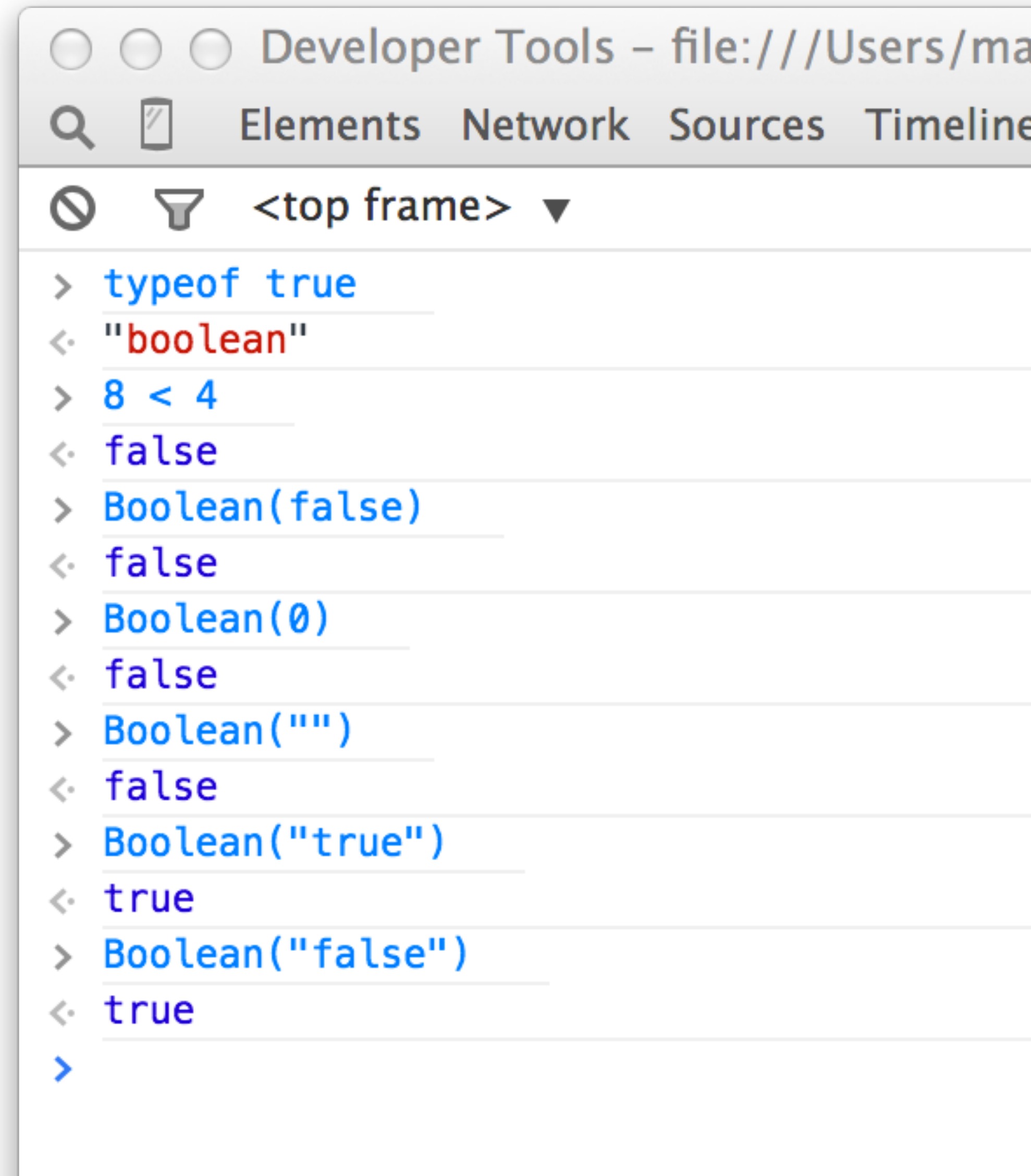
- String variables are also converted to objects as needed.
- `String.toUpperCase()`
- `String.substring(start, end)`
- Note the difference between `.substring()` and `.length`
  - One is a method, one is a property



```
Developer Tools - http://www.cs.arizo
Elements Network Sources Timeline Pr
<top frame>
> t = "A long time ago..."
< "A long time ago..."
> t.toUpperCase()
< "A LONG TIME AGO..."
> t.substring(7, 11)
< "time"
> typeof t
< "string"
> t.length
< 18
>
```

# Boolean

- Boolean for `true` and `false`.
- Comparisons
- Coerce other datatypes into Boolean.
- Note the behavior of the Boolean value for strings.
  - Empty string is `false`
  - Other strings are `true`. Even “false”!



```
Developer Tools – file:///Users/ma
Elements Network Sources Timeline
<top frame>
> typeof true
< "boolean"
> 8 < 4
< false
> Boolean(false)
< false
> Boolean(0)
< false
> Boolean("")
< false
> Boolean("true")
< true
> Boolean("false")
< true
>
```

# Variables

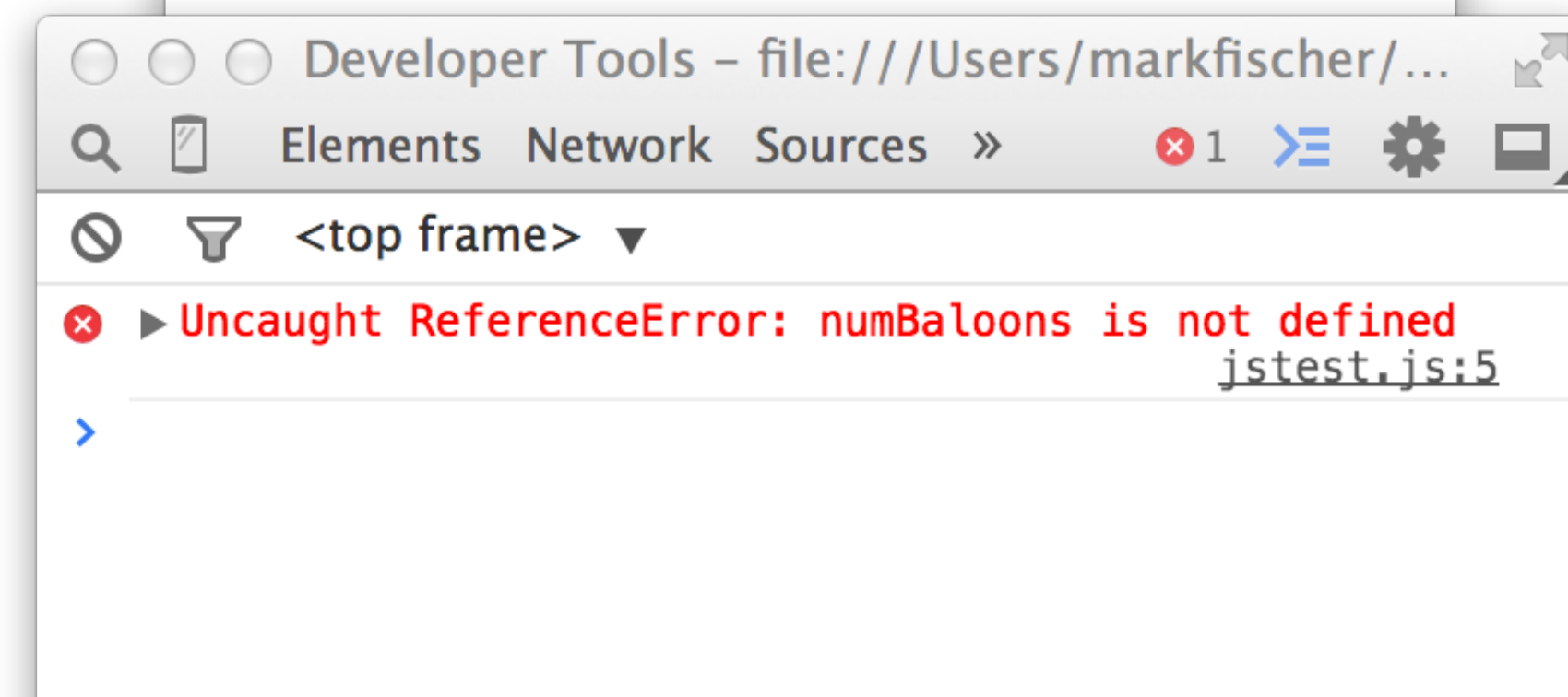
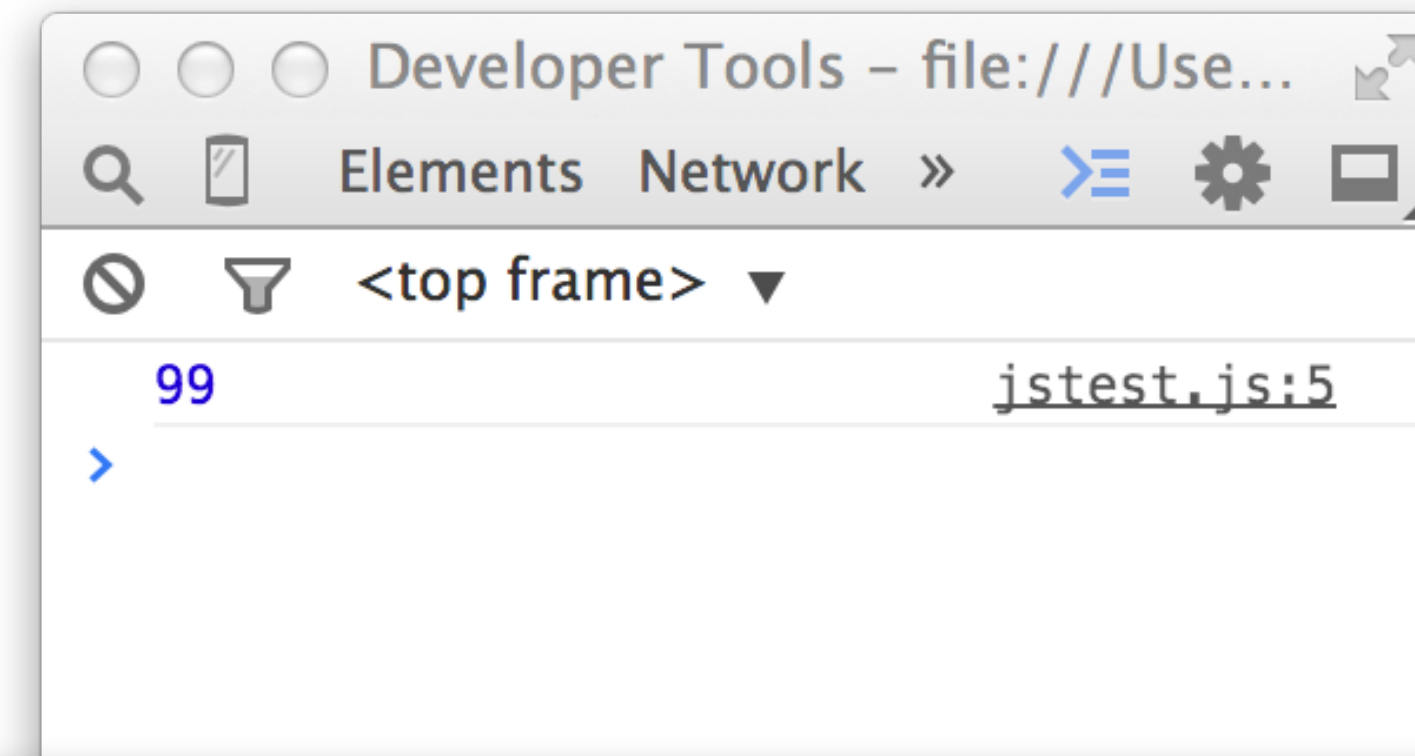
- Variable names can be any combination of letters, numbers, an underscore (`_`), or `$`
- Variable names cannot start with a number.
- Variables do not need to be declared.
- The `var` keyword can be used to declare and scope variables.

# Variables

- Variables have global scope unless `var` is used to declare a variable.

```
var foo = function() {  
  numBalloons = 99;  
}  
foo();  
console.log(numBalloons);
```

```
var foo = function() {  
  var numBalloons = 99;  
}  
foo();  
console.log(numBalloons);
```



# Arrays

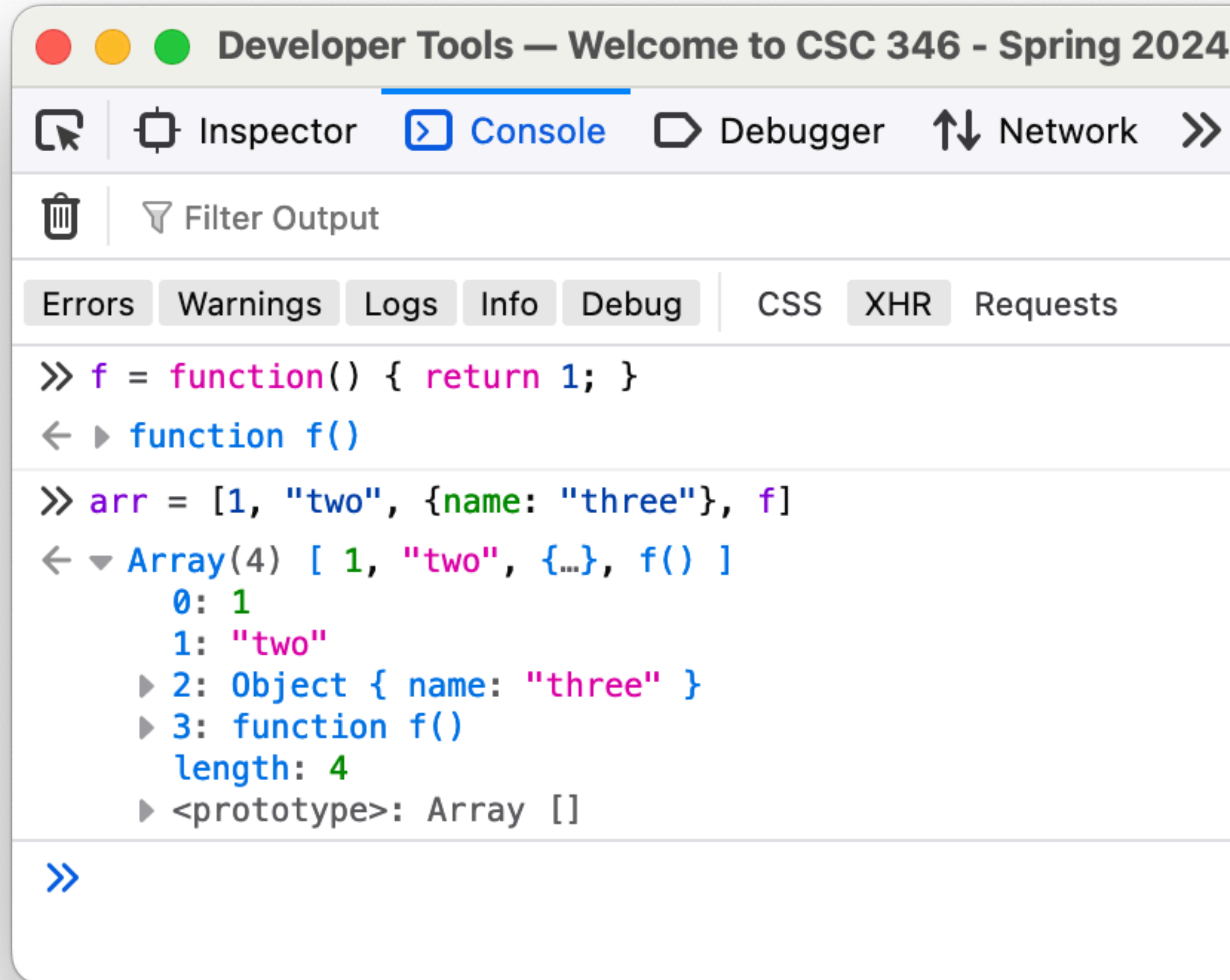
- Collection of values
- Created with `[n, n+1, ...k-1]` syntax
- Array access with brackets: `n[ ]`
- Length property
- Standard Zero based indexing

```
Developer Tools - http://www.cs.arizona.edu
Elements Network Sources Timeline Profiles
<top frame>
> arr = [1, 1, 2, 3, 5, 8];
< [1, 1, 2, 3, 5, 8]
> arr[0]
< 1
> arr.length
< 6
> arr[6]
< undefined
> arr[5]
< 8
>
```



# Arrays

- Arrays can be collections of many different datatypes.



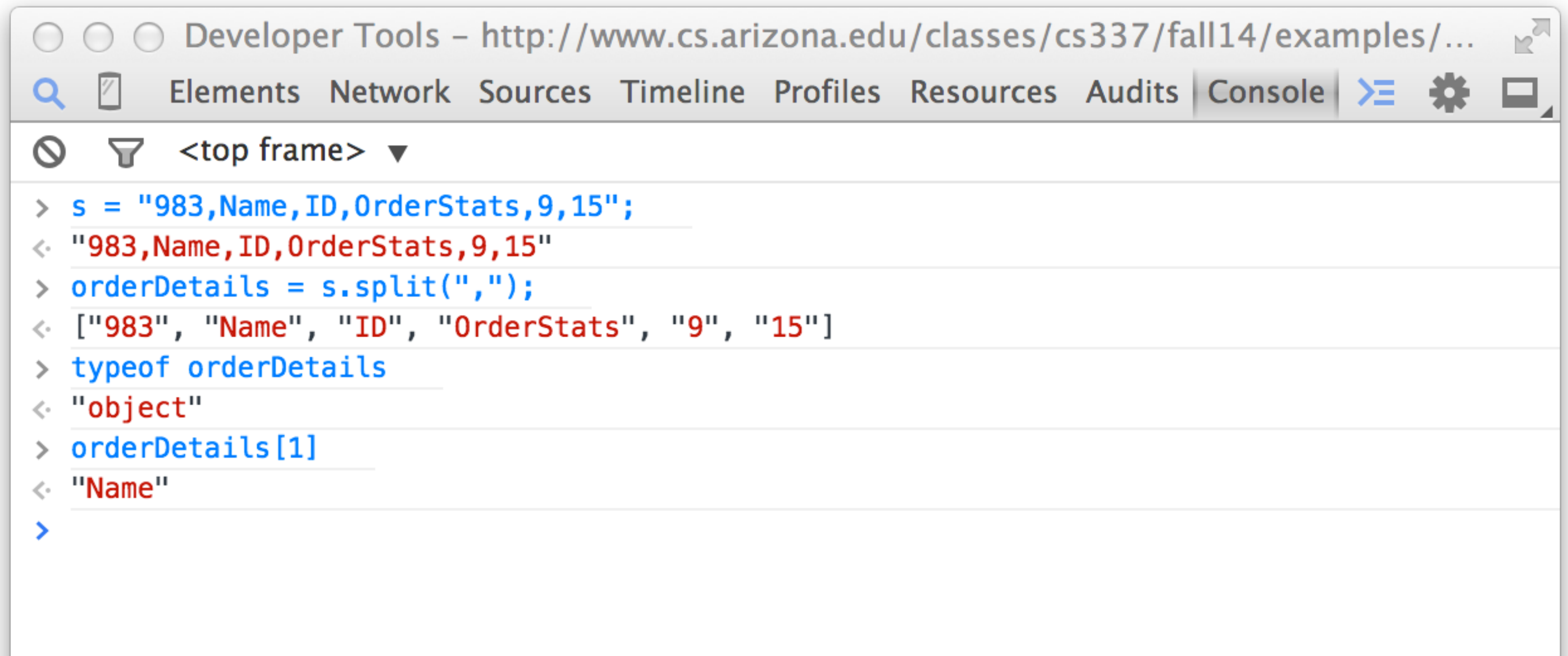
The screenshot shows the Chrome Developer Tools Console with the following content:

```
Developer Tools — Welcome to CSC 346 - Spring 2024
Inspector Console Debugger Network
Filter Output
Errors Warnings Logs Info Debug CSS XHR Requests
>> f = function() { return 1; }
< ▶ function f()
>> arr = [1, "two", {name: "three"}, f]
< ▼ Array(4) [ 1, "two", {...}, f() ]
  0: 1
  1: "two"
  2: Object { name: "three" }
  3: function f()
  length: 4
  <prototype>: Array []
>>
```



# Arrays From Strings

- `String.split()` to create an array from a string.



```
Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examples/...
Elements Network Sources Timeline Profiles Resources Audits Console
<top frame>
> s = "983,Name,ID,OrderStats,9,15";
< "983,Name,ID,OrderStats,9,15"
> orderDetails = s.split(",");
< ["983", "Name", "ID", "OrderStats", "9", "15"]
> typeof orderDetails
< "object"
> orderDetails[1]
< "Name"
>
```

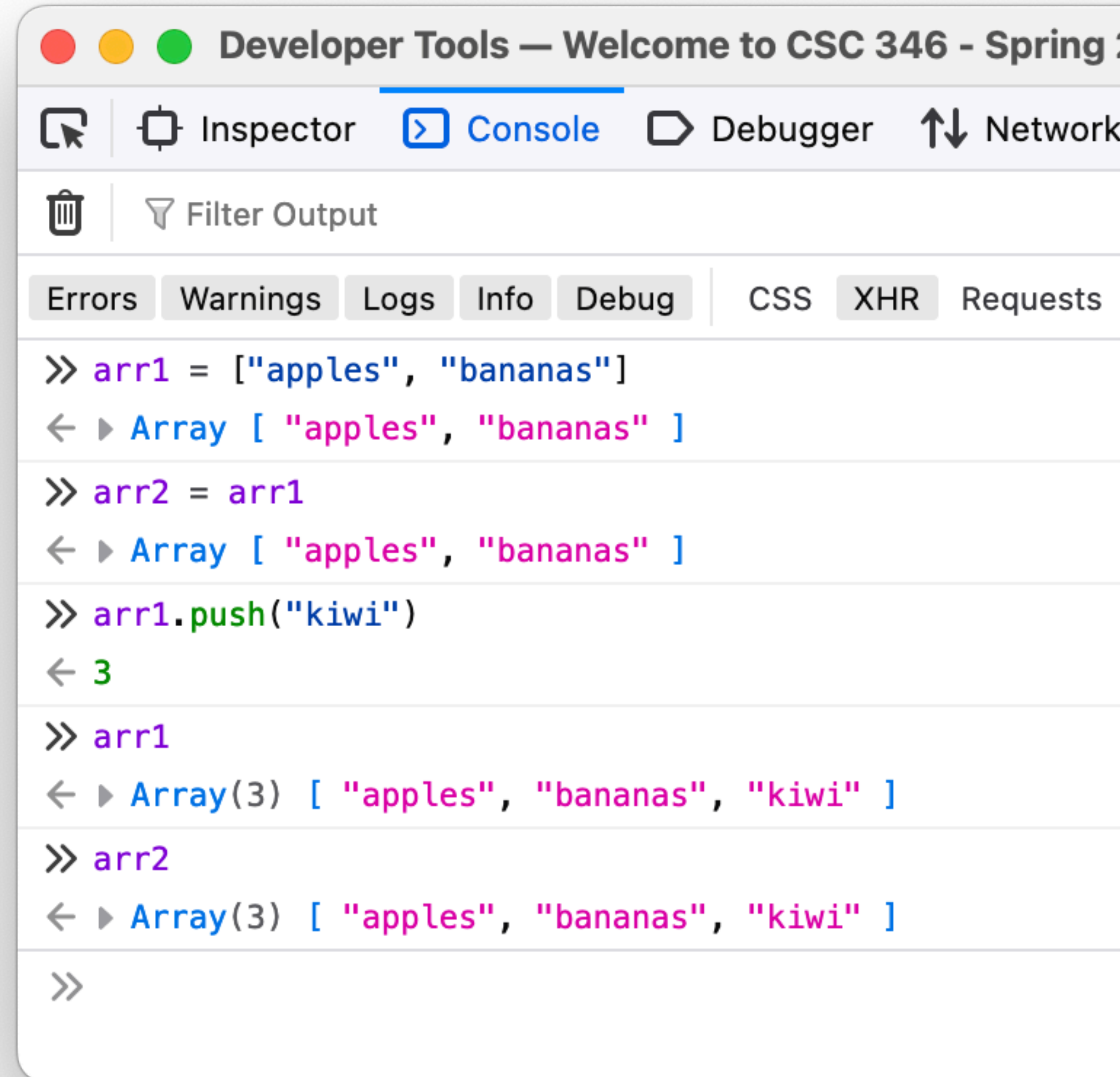
# Array Methods

[https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/Array](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array)

- Lots of useful array methods.
- `.contains(<some value>)` // returns true or false
- `.join(<glue string>)` // joins all elements together with glue and returns a string.
- `.toString()` // Quick string representation of the array
- `.pop()` `.push()` `.shift()` `.unshift()` // Standard array methods
- `.sort()` // Sorts elements according to criteria
- `.splice()` // Adds or removes elements from an array

# Array Assignment

- Assigning an array to another variable assigns a reference of the array to the variable, not a copy.

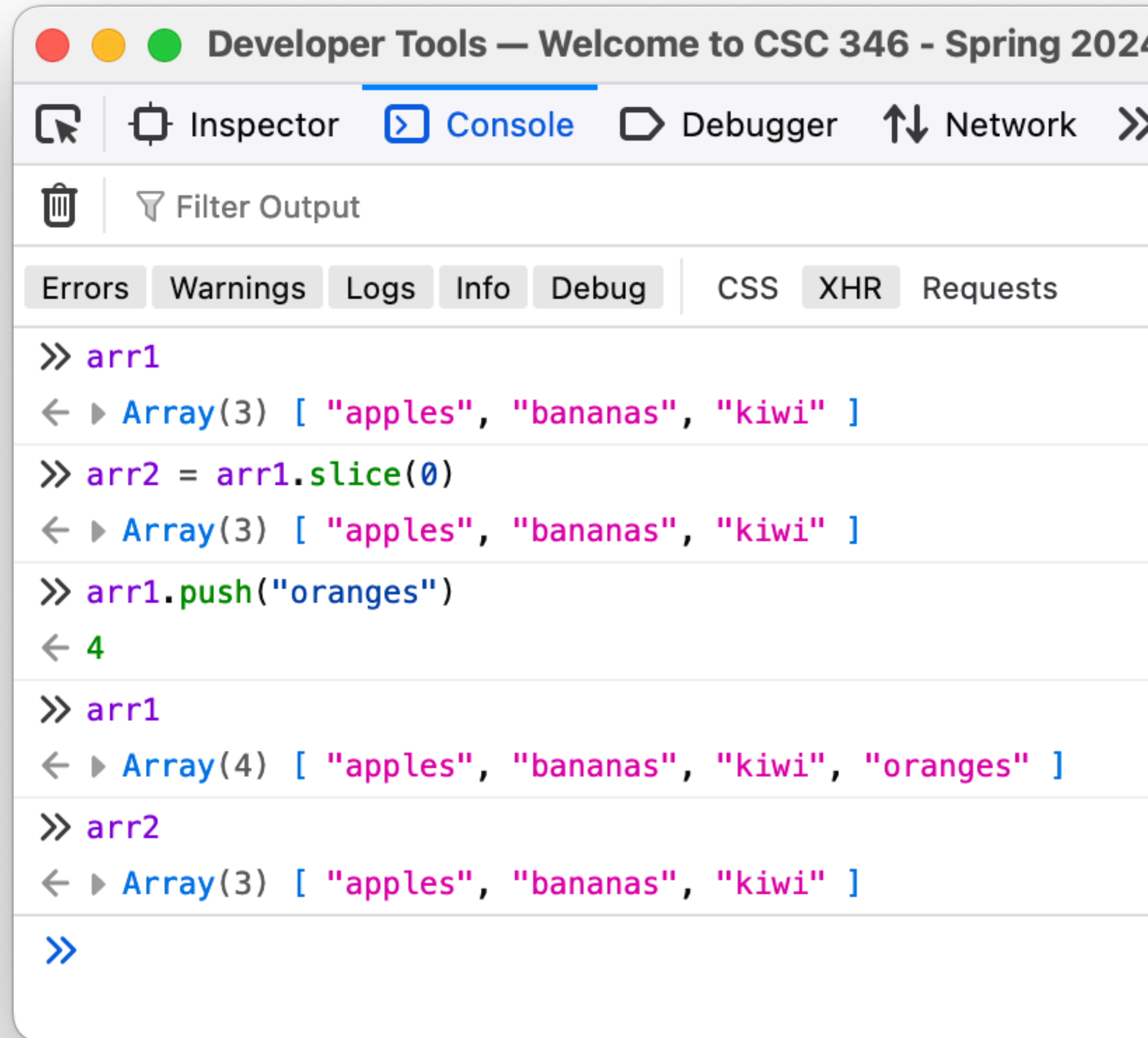


The screenshot shows the Chrome Developer Tools Console with the following content:

```
Developer Tools — Welcome to CSC 346 - Spring 2020
Inspector Console Debugger Network
Filter Output
Errors Warnings Logs Info Debug CSS XHR Requests
>> arr1 = ["apples", "bananas"]
< ▶ Array [ "apples", "bananas" ]
>> arr2 = arr1
< ▶ Array [ "apples", "bananas" ]
>> arr1.push("kiwi")
< 3
>> arr1
< ▶ Array(3) [ "apples", "bananas", "kiwi" ]
>> arr2
< ▶ Array(3) [ "apples", "bananas", "kiwi" ]
>>
```

# Array Assignment

- To make a copy of an array, use the `.slice(0)` method.



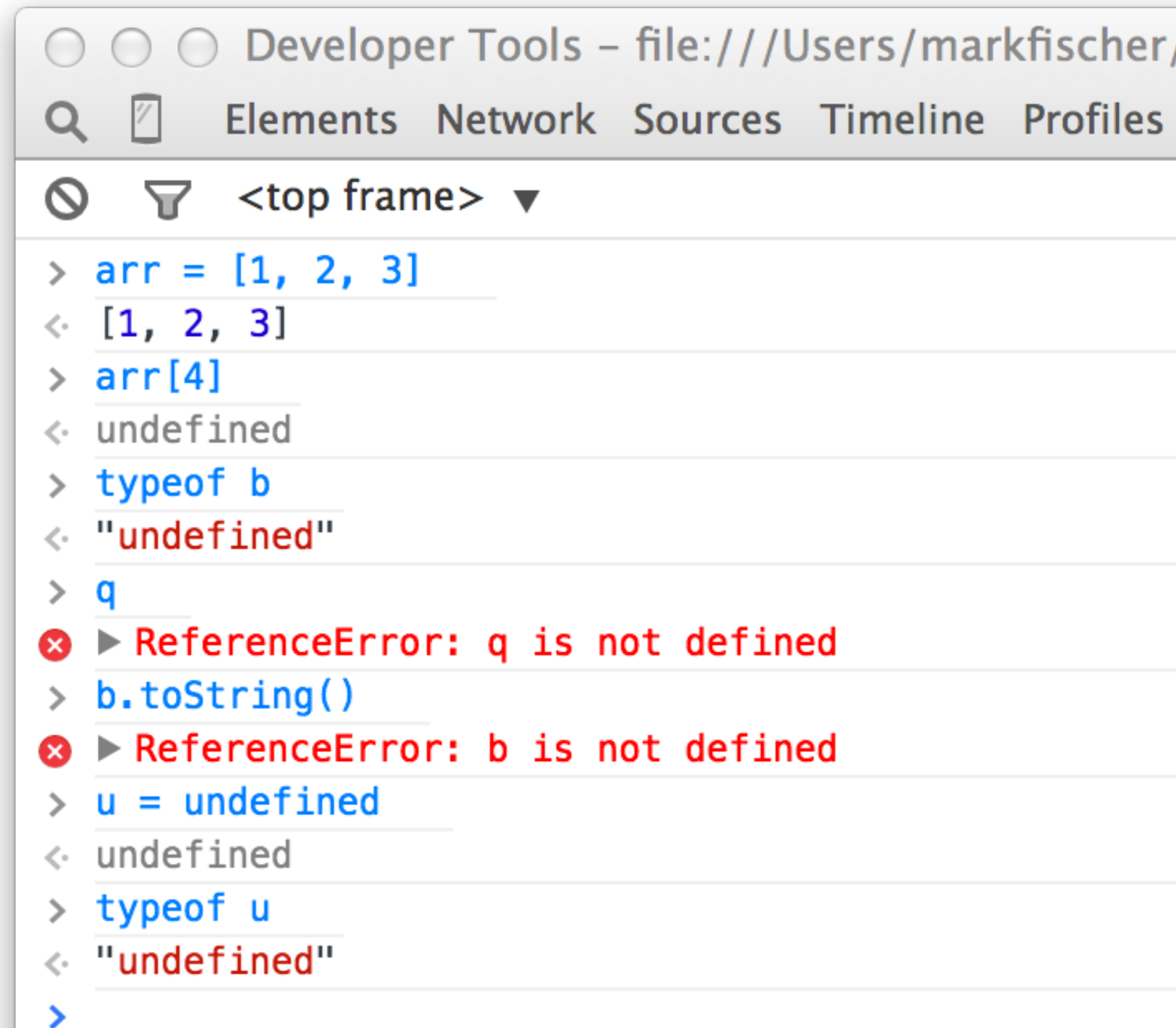
The screenshot shows the developer tools console with the following content:

```
Developer Tools — Welcome to CSC 346 - Spring 2024
Inspector Console Debugger Network
Filter Output
Errors Warnings Logs Info Debug CSS XHR Requests
>> arr1
← ▶ Array(3) [ "apples", "bananas", "kiwi" ]
>> arr2 = arr1.slice(0)
← ▶ Array(3) [ "apples", "bananas", "kiwi" ]
>> arr1.push("oranges")
← 4
>> arr1
← ▶ Array(4) [ "apples", "bananas", "kiwi", "oranges" ]
>> arr2
← ▶ Array(3) [ "apples", "bananas", "kiwi" ]
>>
```

# undefined

[developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/undefined](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/undefined)

- Javascript has a special value for things that are not defined: `undefined`
- Out of bounds requests
- Un-initialized variables
- `undefined` is a property of the *global object*. Its type is `undefined`.



```
Developer Tools - file:///Users/markfischer/
Elements Network Sources Timeline Profiles
<top frame>
> arr = [1, 2, 3]
< [1, 2, 3]
> arr[4]
< undefined
> typeof b
< "undefined"
> q
✖ ▶ ReferenceError: q is not defined
> b.toString()
✖ ▶ ReferenceError: b is not defined
> u = undefined
< undefined
> typeof u
< "undefined"
>
```

# Objects

- Objects are very flexible data structures.
- A basic object:

```
o = {id: 1, name: "an object", counter: 10};
```

- Create property names and values using `key: value` syntax.
- Separate multiple properties by commas.

# Objects

- Access properties via dot syntax

```
o = {id: 1, name: "an object", counter: 10};
```



# Objects

- Act as “Associative Arrays” or “Key / Value” arrays, or “Dictionary” array
- `arr["key"]` syntax

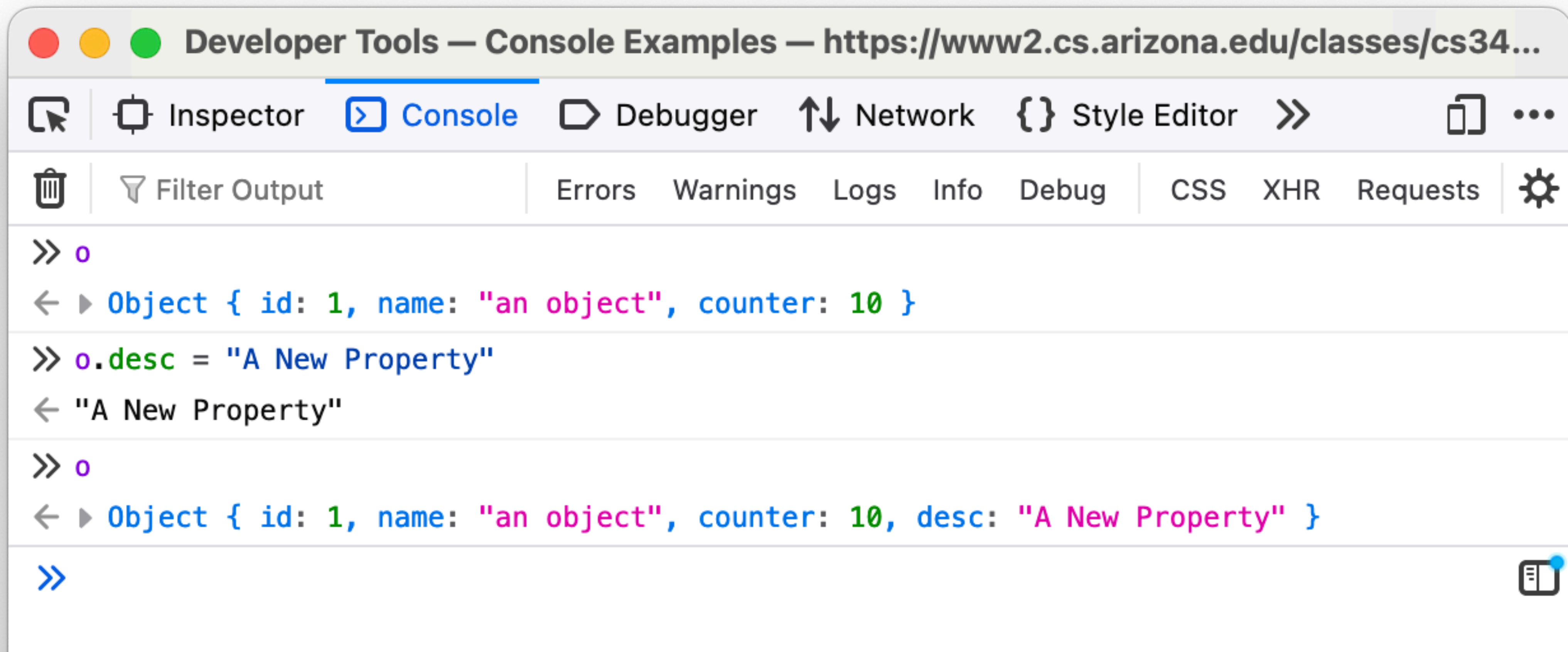
```
o = {id: 1, name: "an object", counter: 10};
```





# Objects

- Assigning to undefined properties creates them



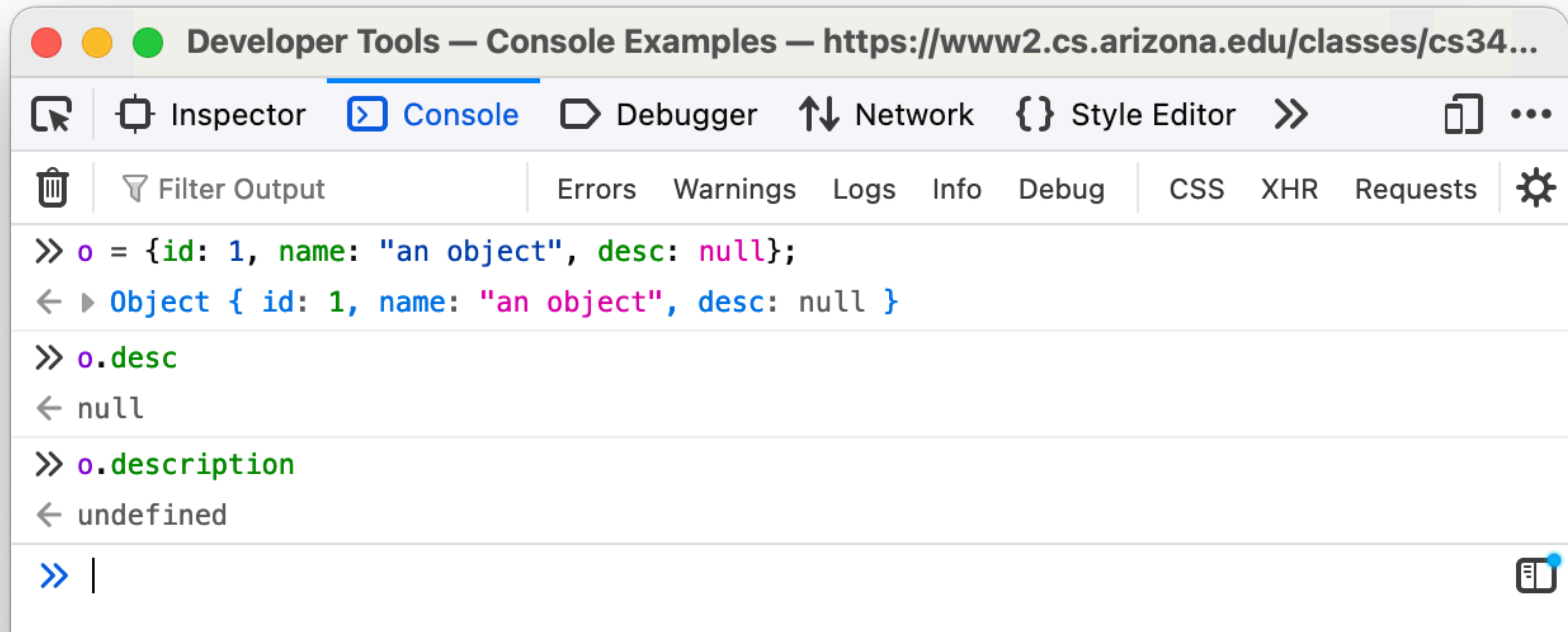
The screenshot shows the Chrome Developer Tools Console. The title bar reads "Developer Tools — Console Examples — https://www2.cs.arizona.edu/classes/cs34...". The "Console" tab is selected. The console output shows the following sequence of commands and results:

```
>> o  
← ▶ Object { id: 1, name: "an object", counter: 10 }  
  
>> o.desc = "A New Property"  
← "A New Property"  
  
>> o  
← ▶ Object { id: 1, name: "an object", counter: 10, desc: "A New Property" }  
  
>>
```

# null

[developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\\_Objects/null](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/null)

- Null is a literal value representing an “empty” or non-existent value.



The screenshot shows a browser window with the title "Developer Tools — Console Examples — https://www2.cs.arizona.edu/classes/cs34...". The "Console" tab is active, displaying the following code and output:

```
>> o = {id: 1, name: "an object", desc: null};  
← Object { id: 1, name: "an object", desc: null }  
  
>> o.desc  
← null  
  
>> o.description  
← undefined  
  
>> |
```

# Operators

- Arithmetic Operators: + - / \* % ++ --
- String concatenation: +
- Logical Operators: && || !
- Comparisons: < > <= >=
- Ternary Operator: condition ? true expr : false expr
- Bitwise Operators: << >> ^ ~

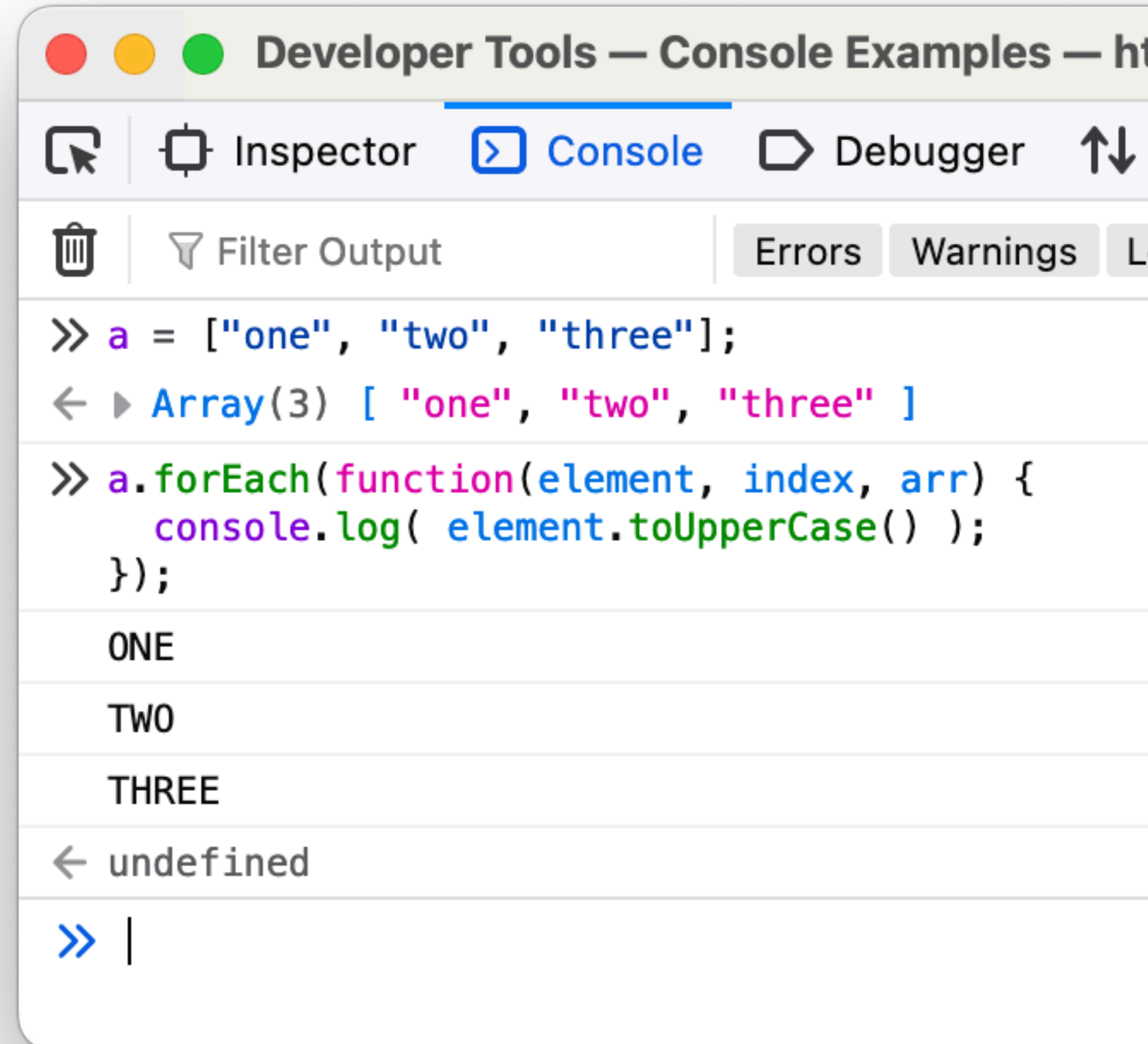
# Control Structures

- `if (condition) { stmt1 } else { stmt2 }`
- `while (condition) { statements }`
- `for (i = 0; i < 10; i++) { statements }`
- Pretty much work like every other C or Java style language

# Control Structures: forEach

- Arrays have a special `forEach` method for performing some action relating to each element of the array
- The `forEach` method takes a *function* as an argument.

```
a = ["one", "two", "three"];
a.forEach(function(element, index, arr) {
  console.log( element.toUpperCase() );
});
```



```
Developer Tools — Console Examples — ht
Inspector Console Debugger
Filter Output Errors Warnings L
>> a = ["one", "two", "three"];
← ▶ Array(3) [ "one", "two", "three" ]
>> a.forEach(function(element, index, arr) {
  console.log( element.toUpperCase() );
});
ONE
TWO
THREE
← undefined
>> |
```

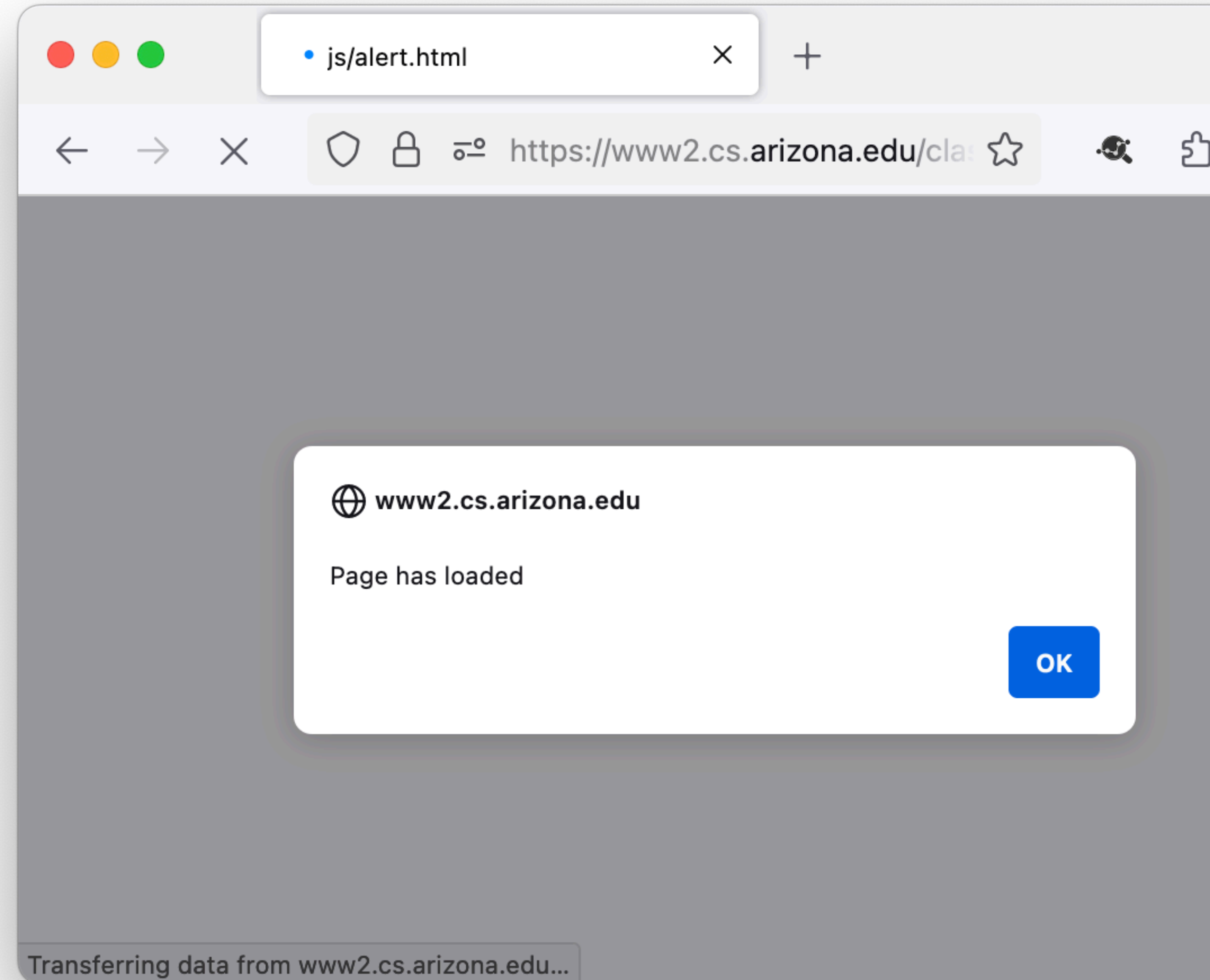
# Basic I/O

- Alerts
- Log to Console
- Confirms
- Prompt
- DOM Manipulation
- Debugger
- No Direct Local File I/O!

# alert ( )

- Display a modal dialog box with the specified text.
- Pauses execution of Javascript until dialog is dismissed.

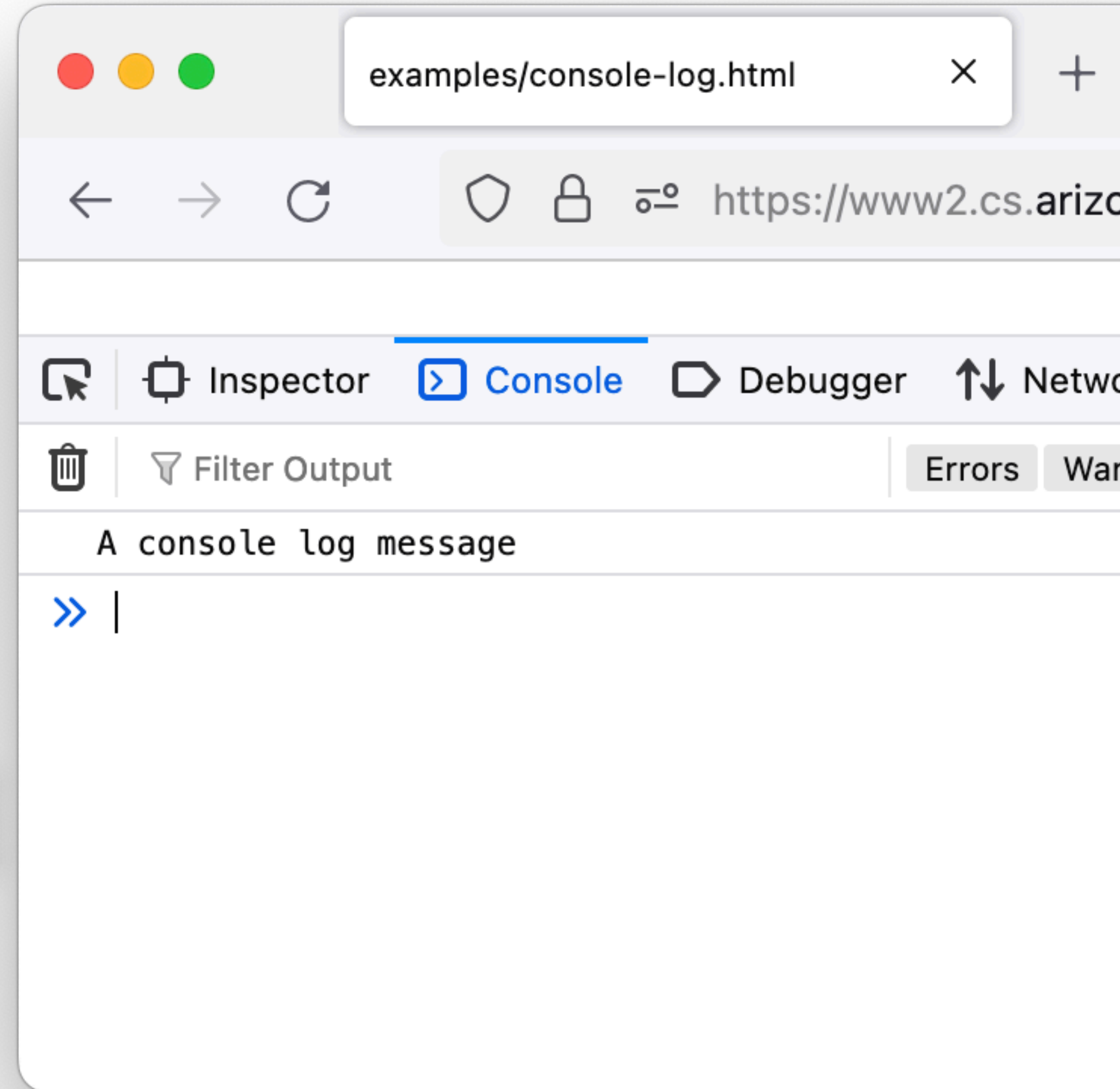
```
alert ("Hello World");
```



# console.log ( )

- Quick way to get some debugging out.
- Doesn't block execution, so usually a better choice for debugging and testing than `alert()`.

```
console.log("A console log message");
```





# Debugger

- Most browsers have a full featured interactive debugger built in.
- Breakpoints, watched expressions, step through execution, etc.
- Example.

The screenshot shows a browser's developer debugger interface. The top navigation bar includes tabs for Inspector, Console, Debugger (active), Network, Style Editor, and Performance. The main editor displays a JavaScript file named 'debugger.js' with the following code:

```
1 var lastFib = 0;
2 var currentFib = 1;
3
4 var nextFib = function() {
5   var nextFib = lastFib + currentFib; nextFib: 3
6   lastFib = currentFib;
7   currentFib = nextFib;
8   return nextFib;
9 }
10
11 for (i = 0; i < 10; i++) {
12   console.log(nextFib());
13 }
14
15
```

The debugger is paused at line 5. The right-hand sidebar shows the following information:

- Paused while stepping** (indicated by a yellow background)
- Watch expressions:** tabContentContainer: (unavailable)
- Breakpoints:** Pause on debugger statement (checked), Pause on exceptions (unchecked). A breakpoint is set on the expression `lastFib + currentFib;` in the file `debugger.js`.
- Call stack:** nextFib (global)
- Scopes:** nextFib scope containing `<this>: Window`, `arguments: Arguments`, and `nextFib: 3`. The global scope is also visible.

# Functions

- Multiple ways to define a function

```
function echo(a) {  
    return a;  
}  
  
echoTwo = function(a) {  
    return a;  
}  
  
var echoThree = function(a) {  
    return a;  
}  
  
console.log( echo("one") );  
console.log( echoTwo("two") );  
console.log( echoThree("three") );
```

# Functions

Declares a named function without requiring assignment



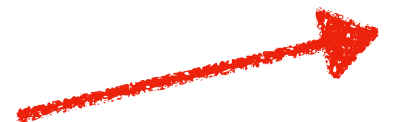
```
function echo(a) {  
  return a;  
}
```

Declares a *global* variable `echoTwo` and assigns an anonymous function to it



```
echoTwo = function(a) {  
  return a;  
}
```

Declares a *local* variable `echoThree` and assigns an anonymous function to it



```
var echoThree = function(a) {  
  return a;  
}
```

```
console.log( echo("one") );  
console.log( echoTwo("two") );  
console.log( echoThree("three") );
```

# Functions

- Does any of this matter?
- What if we call the functions before they're declared?

```
console.log( echo("one") );  
console.log( echoTwo("two") );  
console.log( echoThree("three") );
```

```
function echo(a) {  
    return a;  
}
```

```
echoTwo = function(a) {  
    return a;  
}
```

```
var echoThree = function(a) {  
    return a;  
}
```

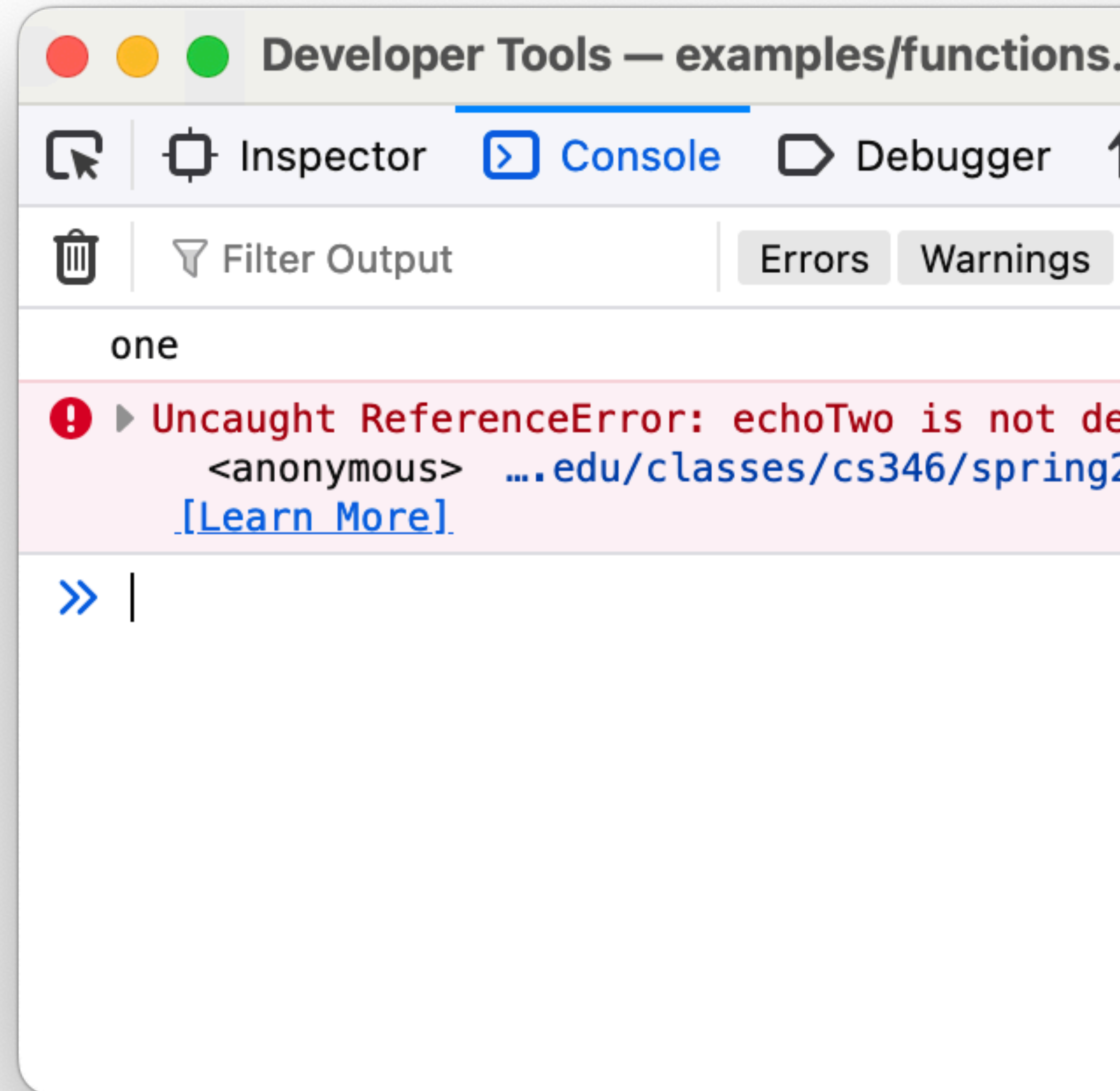
# Functions

```
console.log( echo("one") );
console.log( echoTwo("two") );
console.log( echoThree("three") );

function echo(a) {
  return a;
}

echoTwo = function(a) {
  return a;
}

var echoThree = function(a) {
  return a;
}
```



Developer Tools — examples/functions.

Inspector Console Debugger

Filter Output Errors Warnings

one

! ▶ Uncaught ReferenceError: echoTwo is not de  
    <anonymous> ...edu/classes/cs346/spring2  
    [\[Learn More\]](#)

>> |

# Functions

- The first style has a symbol table entry created for it at parse time. So it can be referenced immediately during runtime.
- The other two have symbol table entries created at runtime, so aren't available until after they've been executed.

```
console.log( echo("one") );  
console.log( echoTwo("two") );  
console.log( echoThree("three") );
```

```
function echo(a) {  
    return a;  
}
```

```
echoTwo = function(a) {  
    return a;  
}
```

```
var echoThree = function(a) {  
    return a;  
}
```

# Functions

- So should we always use Function Declarations?
  - Well, it depends...

```
//Function Declaration  
function add(a,b) {return a + b};  
//Function Expression  
var add = function (a,b) {return a + b};
```

# Functions

- What if we want to re-define a function somewhere in the code?
- What is the console output here?

```
function echo(a) {  
    return a;  
}  
  
console.log( echo("one") );
```

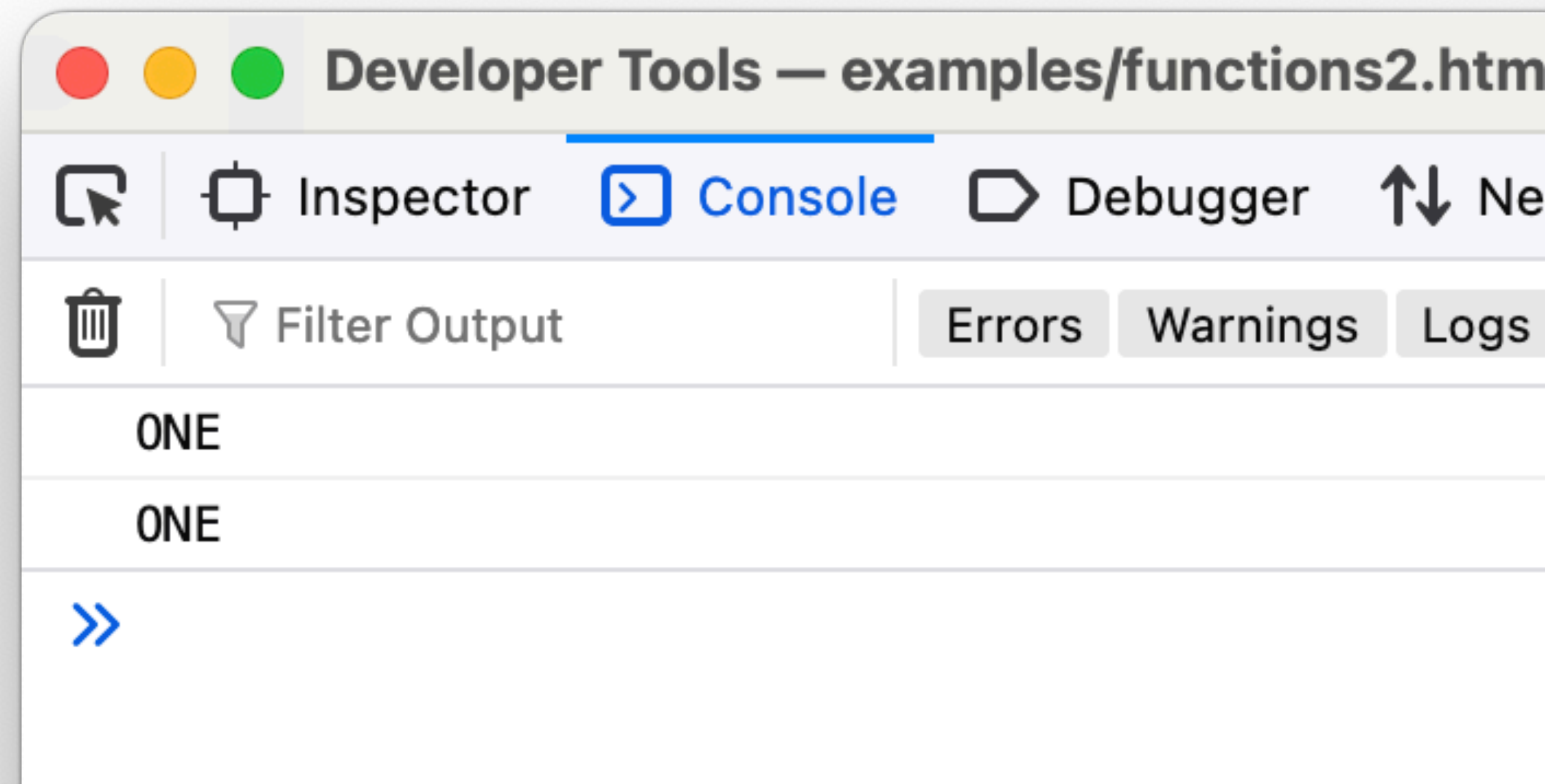
```
function echo(a) {  
    return a.toUpperCase();  
}  
  
console.log( echo("one") );
```



# Functions

- Hmm, maybe not what we were expecting.
- Function Declarations are 'hoisted' to the top at parse time, so when executed, the last declared version wins.

```
function echo(a) {  
    return a;  
}  
  
console.log( echo("one") );  
  
function echo(a) {  
    return a.toUpperCase();  
}  
  
console.log( echo("one") );
```



# Function Declarations

- Can only appear as block level elements.
- Are 'hoisted' to the top at parse time, before run time.
- Cannot be nested within non-function blocks.
- Are scoped by where they are declared, like `var`



# Function Expressions

- Can be used anywhere an expression is valid.
  - Can be more flexible because of this.
- Are evaluated and assigned at run time.

# Function Expressions

- Recall that functions are first-class data types in JavaScript. This means that anywhere in the language you can use an expression, you can substitute a function.

```
function logWithFormat(message, formatter) {  
  let formattedMessage = formatter(message)  
  console.log(formattedMessage)  
}
```

```
logWithFormat("Hello", function(s) { return s.toUpperCase() })  
// prints "HELLO" to the console
```

# Arrow Functions

- Many of these “callback” style functions require the same format.
- Of course programmers developed a shorter way to write them

```
logWithFormat("Hello", function(s) { return s.toUpperCase() })
```

```
logWithFormat("Hello", (s) => s.toUpperCase())
```

# Arrow Functions

- Arrow functions are anonymous. There's no named symbol to reference anywhere else
- If the function body is simple, you can omit the `{ }` and the statement will automatically be returned
- If your function body is more complex, you must explicitly return a value
- Arrow functions have other benefits

```
(s) => s.toUpperCase()  
  
(s) => {  
    s2 doWork(s)  
    return s2.toUpperCase()  
}
```

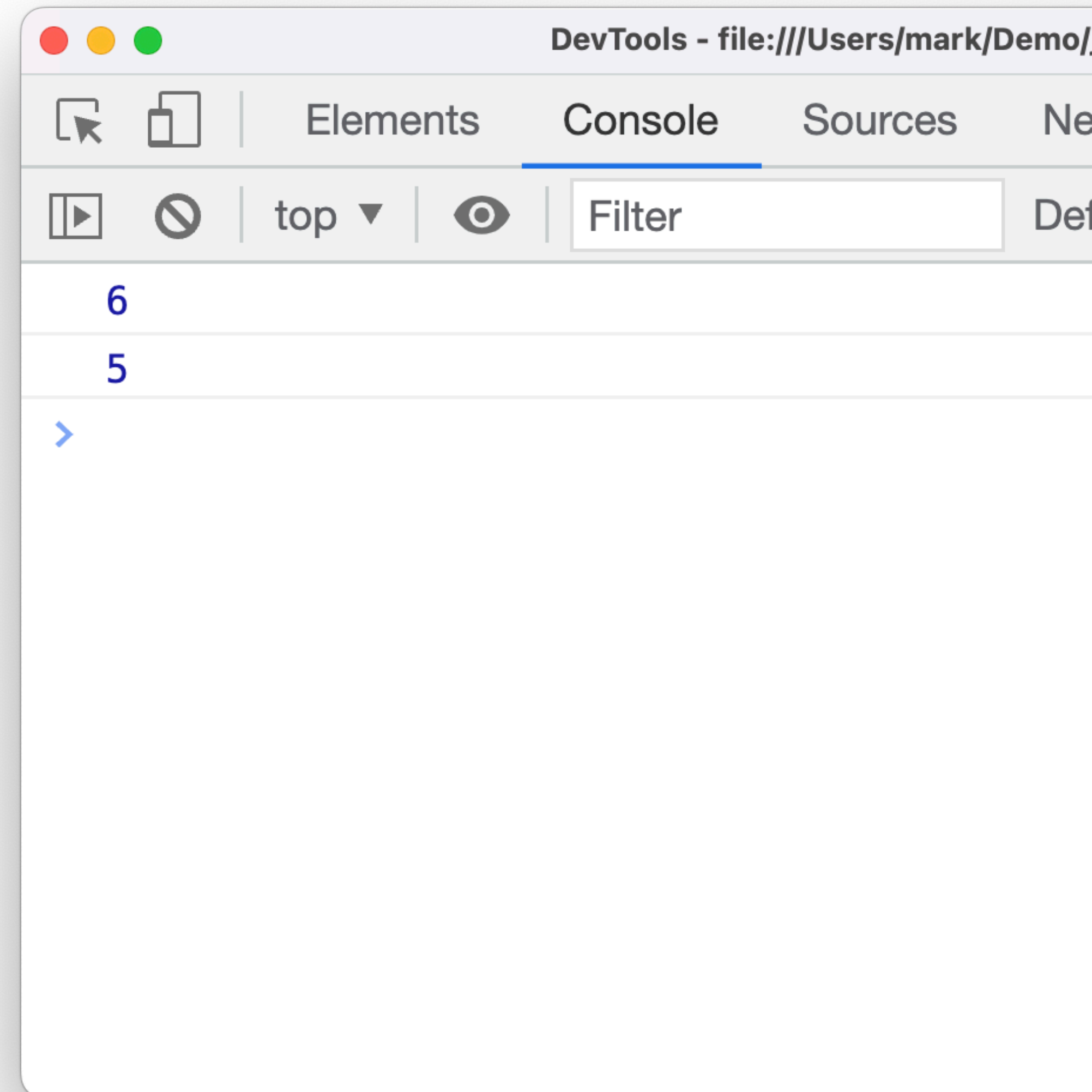
# Objects and Functions

- Functions can be added to objects as property variables.
- Object “methods” are really properties with functions assigned to them.



# Objects and Functions

```
var doubleMe = function(x) {  
  return 2 * x;  
}  
  
var halveMe = function(x) {  
  return x/2;  
}  
  
var myLib = {  
  version: 0.3,  
  name: "My Test Library",  
  double: doubleMe,  
  half: halveMe  
}  
  
console.log( myLib.double(3) );  
console.log( myLib.half(10) );
```





# Objects and Functions

- Using anonymous function expressions instead.
- Arrow functions are especially popular in this situation.

```
var myLib = {  
  version: 0.4,  
  name: "My Test Library",  
  double: function(x) { return 2 * x; },  
  half: function(x) { return x/2; }  
}
```

```
console.log( myLib.double(3) );  
console.log( myLib.half(10) );
```

```
var myLib = {  
  version: 0.4,  
  name: "My Test Library",  
  double: (x) => 2 * x,  
  half: (x) => x/2  
}
```

```
console.log( myLib.double(3) );  
console.log( myLib.half(10) );
```

# Javascript in HTML

- Where does our Javascript live?
- Inline in an HTML document inside a `<script>` element
- Included in an external file via a `<script>` element.

# Javascript in HTML

- The `<script>` element with inline content
- Within the `<script>` element, we're parsing Javascript, not HTML

```
<!doctype html>
<head>
  <title>js/jstest.html</title>

  <script>
    var answer = 42;
    function calculateAnswer() {
      return answer;
    }
    console.log( calculateAnswer() );
  </script>
</head>

<body>
  <div></div>
  <div></div>
</body>
</html>
```

# Javascript in HTML

- The `<script>` element with `src` attribute.
- Includes an external file with Javascript in it.
- No wrapping `<script>` tags within external files.

```
<!doctype html>
<html>
<head>
  <title>js/jstest.html</title>
  <script src="jstest.js"></script>
</head>

<body>
  <div></div>
</body>
</html>
```

```
var answer = 42;
function calculateAnswer() {
  return answer;
}
console.log( calculateAnswer() );
```



# The document Object

**This is all well and good, but how about something involving a web page?**

# The document Object

- The `document` object is ***NOT*** part of the Javascript language.
- You won't find it in server contexts such as `node.js` for example.
- It is an API defined by the W3C to interact with HTML and XML documents.

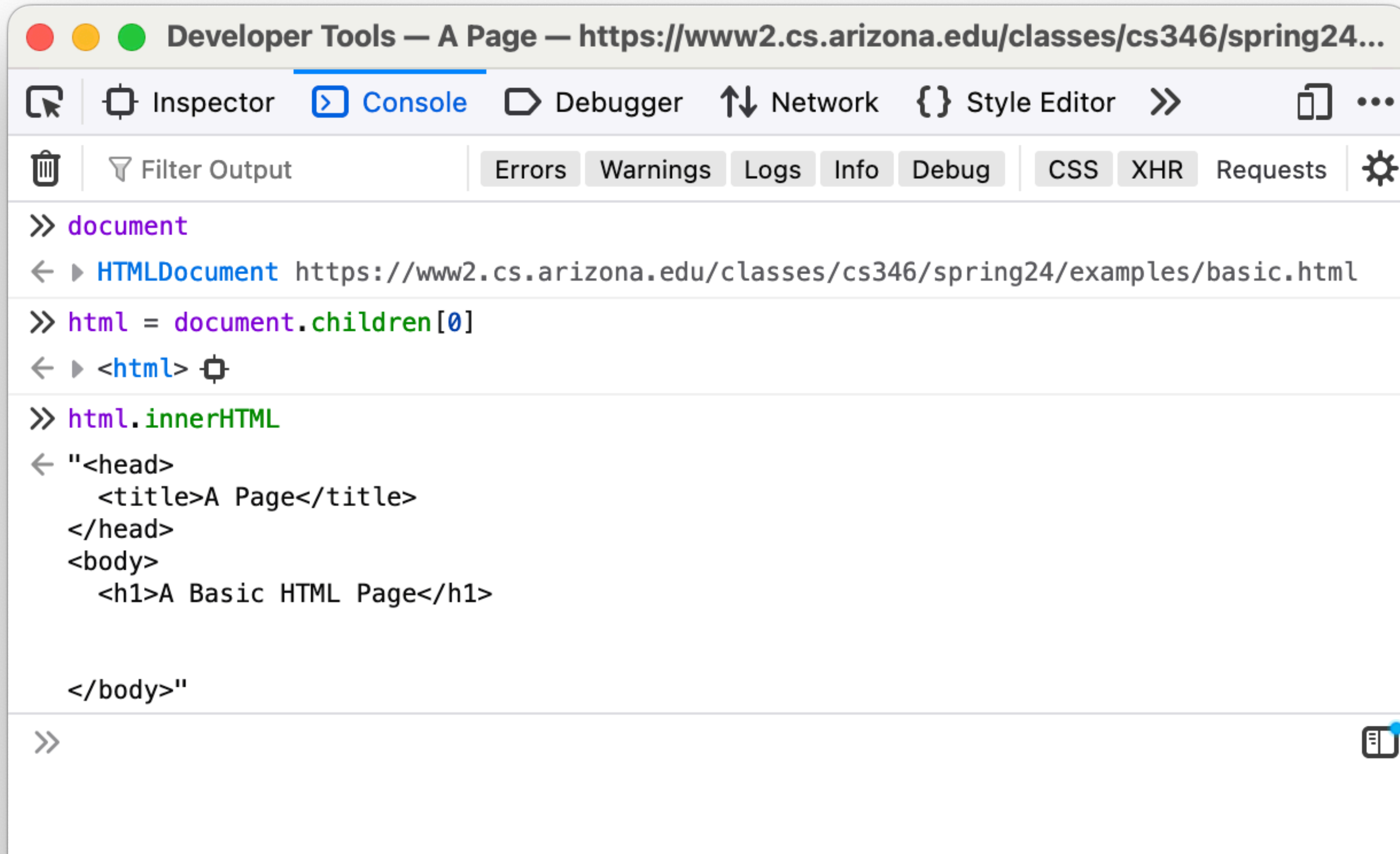
[https://developer.mozilla.org/en-US/docs/Web/API/Document\\_Object\\_Model](https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model)



# The document Object

- Browsers parse the HTML and CSS of a page, and build an object model in memory.
- The browser exposes this object to us for use with our Javascript as the `document` object.

# The document Object



The screenshot shows a web browser window with the title "Developer Tools — A Page — https://www2.cs.arizona.edu/classes/cs346/spring24...". The developer tools are open, and the "Console" tab is selected. The console shows the following code and output:

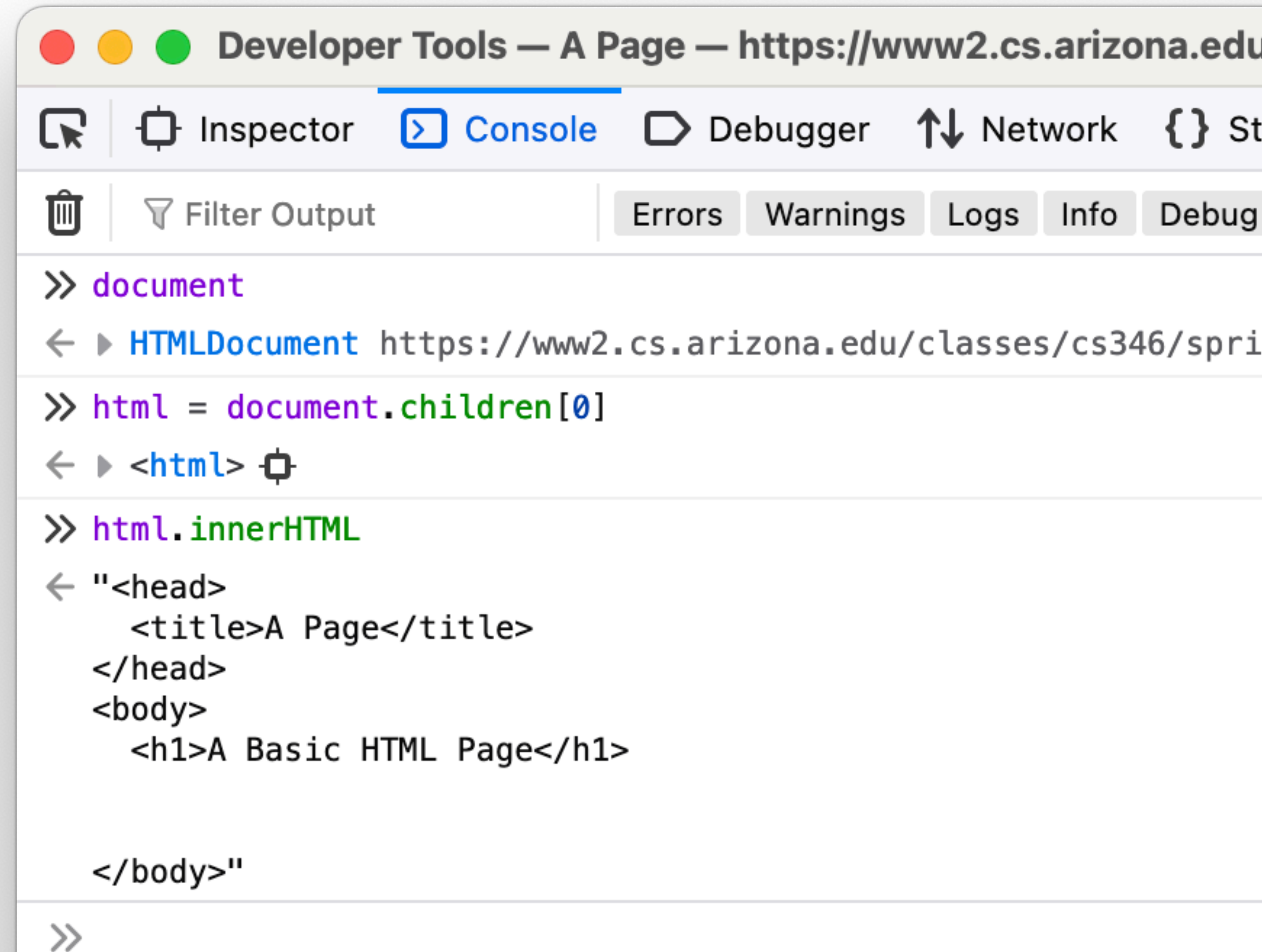
```
>> document
< HTMLDocument https://www2.cs.arizona.edu/classes/cs346/spring24/examples/basic.html
>> html = document.children[0]
< <html>
>> html.innerHTML
< "<head>
  <title>A Page</title>
</head>
<body>
  <h1>A Basic HTML Page</h1>
</body>"
```

The console interface includes a "Filter Output" section with tabs for "Errors", "Warnings", "Logs", "Info", and "Debug". There are also tabs for "CSS", "XHR", and "Requests". A "Settings" gear icon is visible on the right side of the console.



# The document Object

- `document` elements are *objects*, so accessing their properties is done with the dot syntax
- `object.property`
- `html.innerHTML` for example



The screenshot shows a browser window with the title "Developer Tools — A Page — https://www2.cs.arizona.edu". The "Console" tab is active, displaying the following code and output:

```
>> document
< HTMLDocument https://www2.cs.arizona.edu/classes/cs346/spri

>> html = document.children[0]
< <html>

>> html.innerHTML
" <head>
  <title>A Page</title>
</head>
<body>
  <h1>A Basic HTML Page</h1>

</body>"

>>
```

# DOM Selection

- Starting with the `document` root and drilling down via `.children` is tedious. Can we get at elements some other way?
- `document.getElementById("main")`
- `document.getElementsByTagName("p")`
- `document.getElementsByClassName("error")`

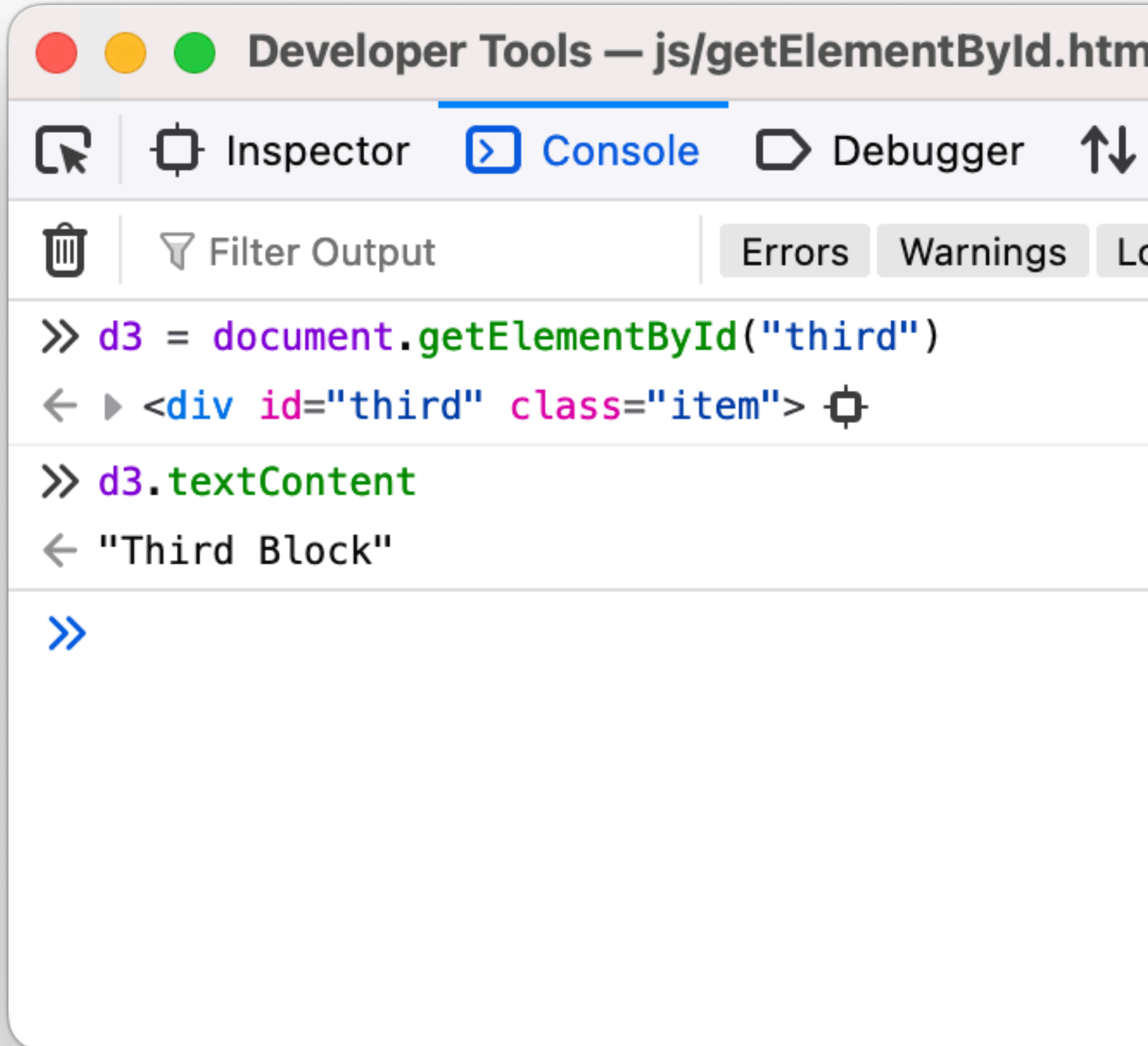
# getElementById

- Gets an HTML element object from the document based on an ID.
- Since ID must be unique, this method returns a single element, not an array of elements.

# getElementById

```
<!doctype html>
<head>
  <title>js/getElementById.html</title>
</head>

<body>
  <div id="main">
    <div id="first" class="item">
      First Block
    </div>
    <div id="second" class="item">
      Second Block
    </div>
    <div id="third" class="item selected">
      Third Block
    </div>
  </div>
</body>
</html>
```



Developer Tools — js/getElementById.html

Inspector Console Debugger

Filter Output Errors Warnings

```
>> d3 = document.getElementById("third")
< div id="third" class="item" >
```

```
>> d3.textContent
"Third Block"
```

```
>>
```

# Updating the DOM

- Now that we can get an element, can we do something with it?

```
<!doctype html>
<head>
  <title>js/getElementById.html</title>
  <link rel="stylesheet" type="text/css"
        href="getElements.css" />
  <script src="getElementById.js"></script>
</head>

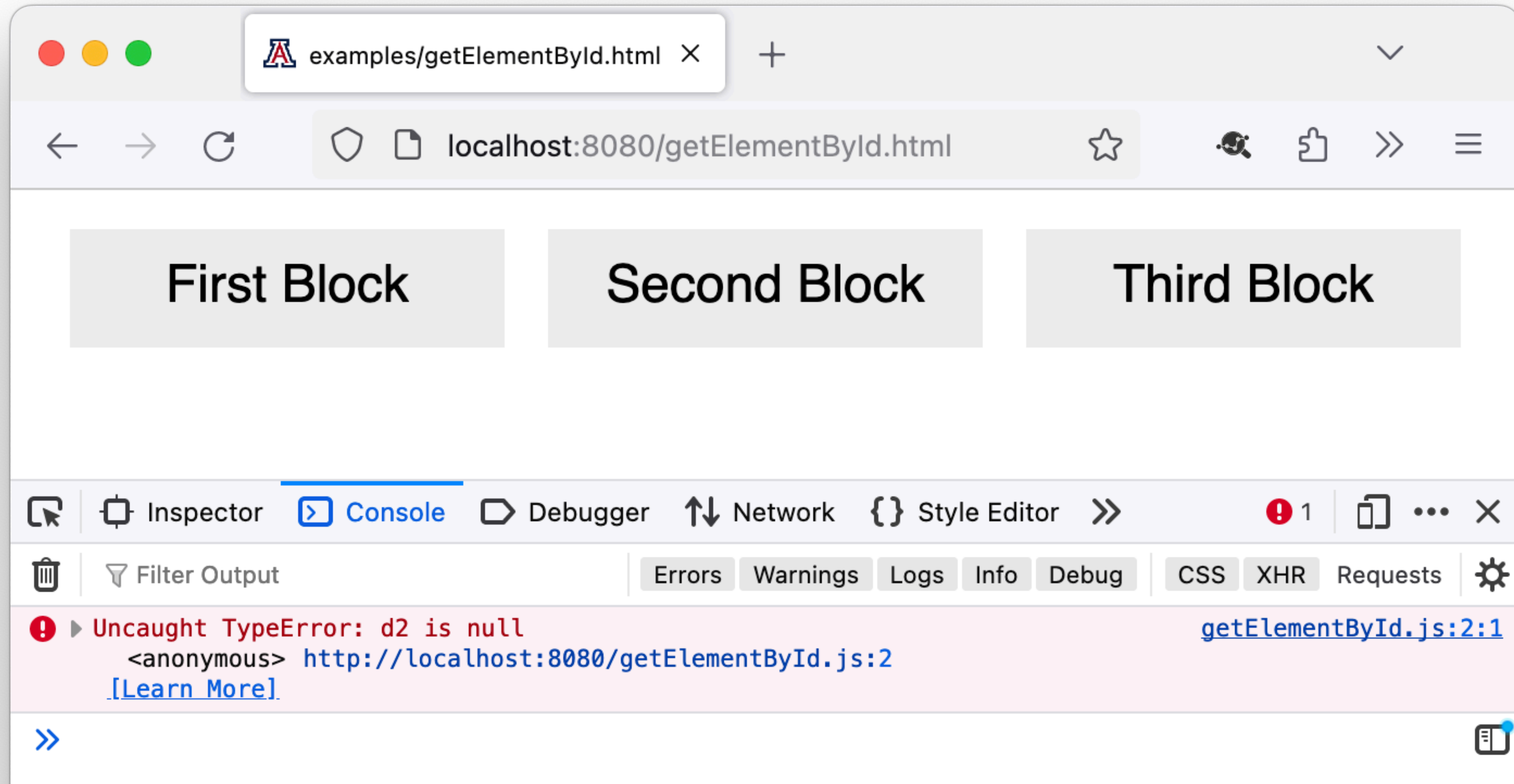
<body>
  <div id="main">
    <div id="first" class="item">First Block</div>
    <div id="second" class="item">Second Block</div>
    <div id="third" class="item">Third Block</div>
  </div>
</body>
</html>
```

```
d2 = document.getElementById('second');
d2.classList.add("selected");
```



# Updating the DOM

- Hmm nothing happened. Why? Check the console.



# Updating the DOM

- Uncaught TypeError: Cannot read properties of null?? But how can d2 be null?

```
d2 = document.getElementById('second');  
d2.classList.add("selected");
```

```
<!doctype html>  
<head>  
  <title>js/getElementById.html</title>  
  <link rel="stylesheet" type="text/css"  
        href="getElements.css" />  
  <script src="getElementById.js"></script>  
</head>  
  
<body>  
  <div id="main">  
    <div id="first" class="item">First Block</div>  
    <div id="second" class="item">Second Block</div>  
    <div id="third" class="item">Third Block</div>  
  </div>  
</body>  
</html>
```

# Waiting for the DOM to load

- The browser waits for no DOM
- The browser parses the file, loads the `getElementById.js` file, and executes it all before the rest of the HTML is parsed and the DOM is created.

```
<!doctype html>
<head>
  <title>js/getElementById.html</title>
  <link rel="stylesheet" type="text/css"
        href="getElements.css" />
  <script src="getElementById.js"></script>
</head>

<body>
  <div id="main">
    <div id="first" class="item">First Block</div>
    <div id="second" class="item">Second Block</div>
    <div id="third" class="item">Third Block</div>
  </div>
</body>
</html>
```





# Waiting for the DOM to load

- What if we just move the `<script>` element down to the bottom?

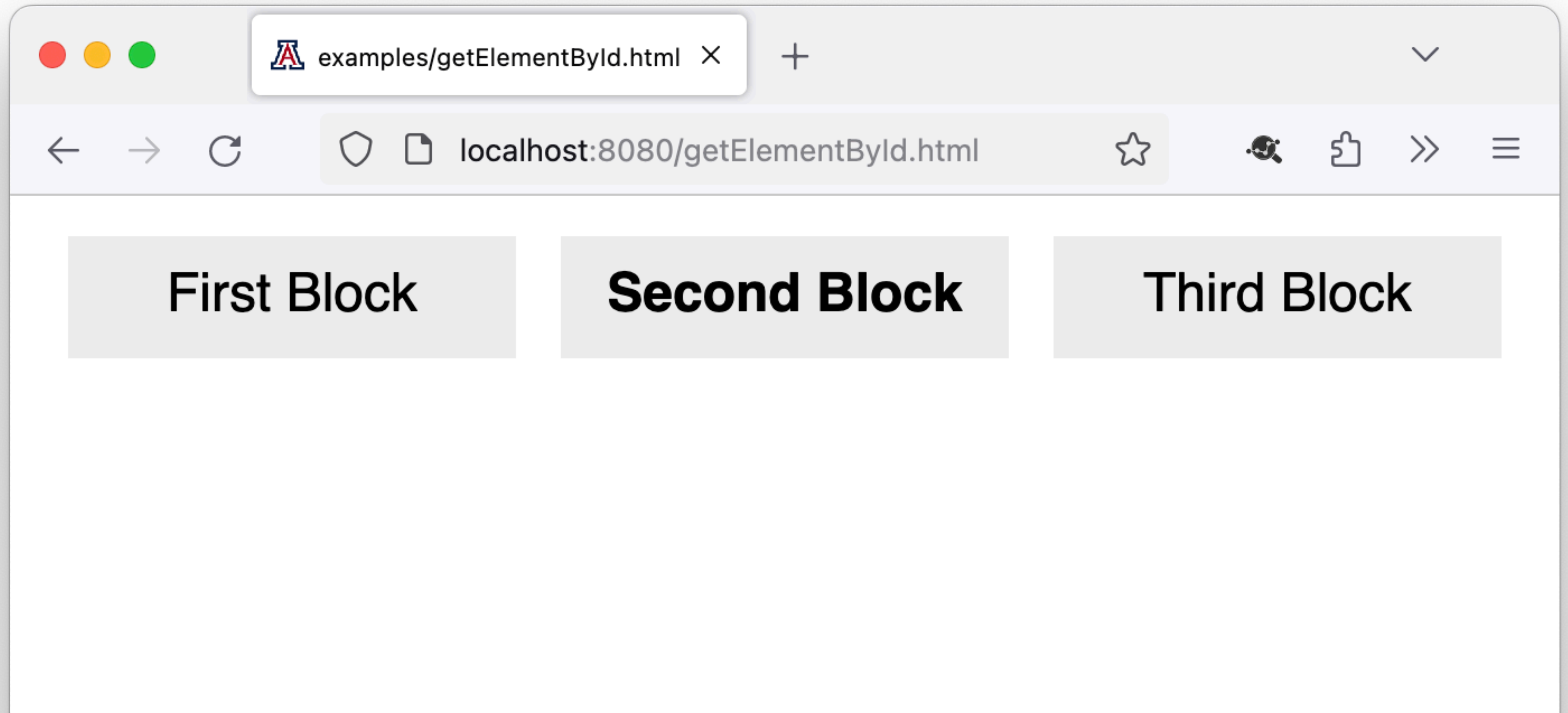
```
<!doctype html>
<head>
  <title>js/getElementById.html</title>
  <link rel="stylesheet" type="text/css"
        href="getElements.css" />
</head>

<body>
  <div id="main">
    <div id="first" class="item">First Block</div>
    <div id="second" class="item">Second Block</div>
    <div id="third" class="item">Third Block</div>
  </div>

  <script src="getElementById.js"></script>
</body>
</html>
```

# Waiting for the DOM to load

- Works!



# Waiting for the DOM to load

- That seems... hackish. Isn't there a "right" way to do this?
- Well, its perfectly valid. `<script>` elements do not have to go in the `<head>`, although they frequently do.
- However, `<script>` elements that aren't in the `<head>` tend to get overlooked later, so we try to put them there if we can.

# Events

- The web browser is an Event Driven application.
- Documents load, links are clicked, HTTP requests are made and completed.
- Each of these is an event, and we can register event listeners (functions) which will be called as these events occur.
- These are called *callbacks*.

# Events

- `object.addEventListener('event', callback);`
- The object can be any object that responds to event listeners, such as an Element, the Document, or maybe the Window.

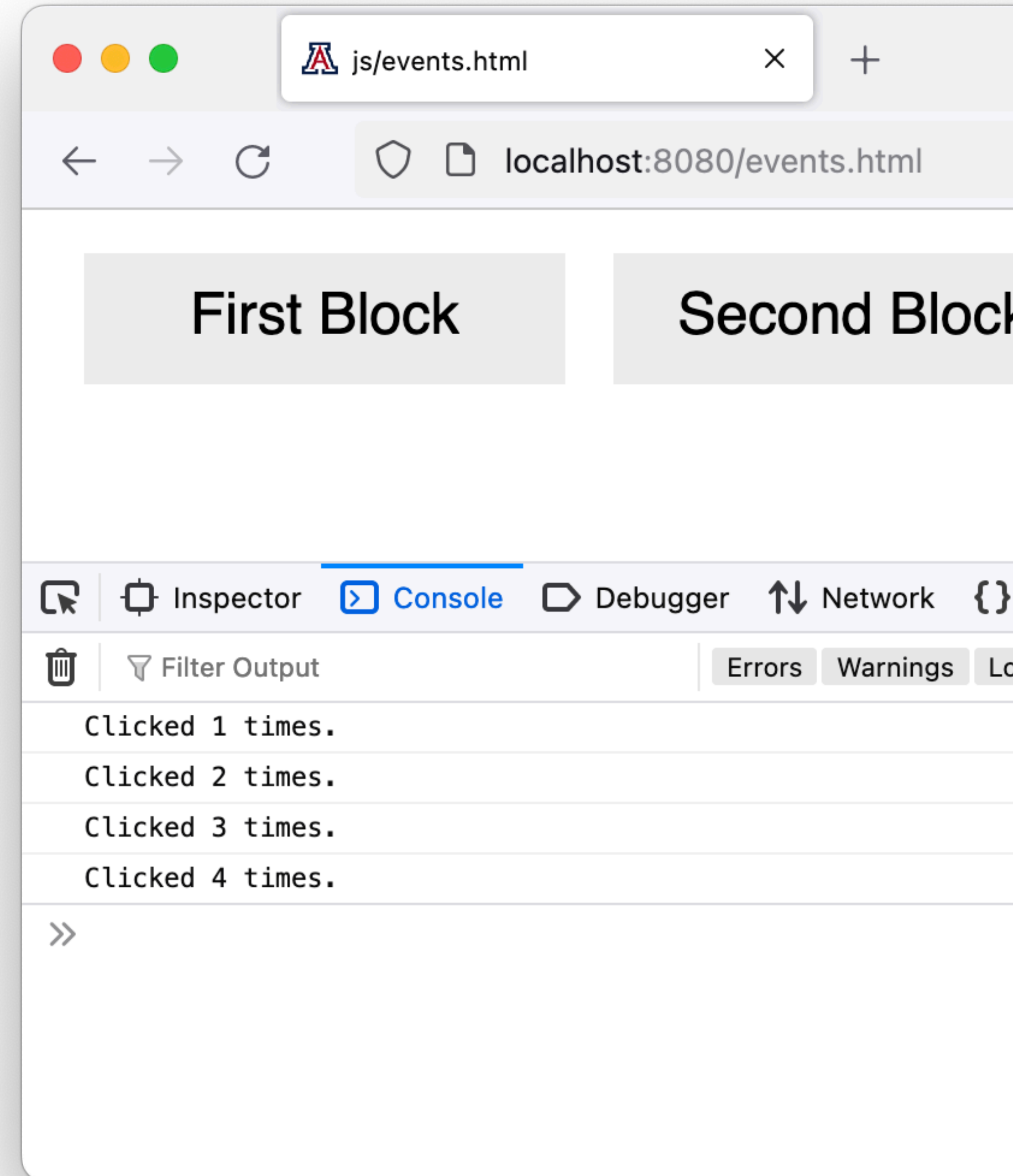
# Events

- A basic example of a 'click' event handler.

```
<!doctype html>
<head>
  <title>js/events.html</title>
  <link rel="stylesheet" type="text/css"
        href="getElements.css" />
</head>

<body>
  <div id="main">
    <div id="first" class="item">First Block</div>
    <div id="second" class="item">Second Block</div>
    <div id="third" class="item">Third Block</div>
  </div>

  <script>
    clickCount = 0;
    d1 = document.getElementById('first');
    d1.addEventListener('click', function() {
      console.log("Clicked " + ++clickCount + " times.");
    });
  </script>
</body>
</html>
```



# window load Event

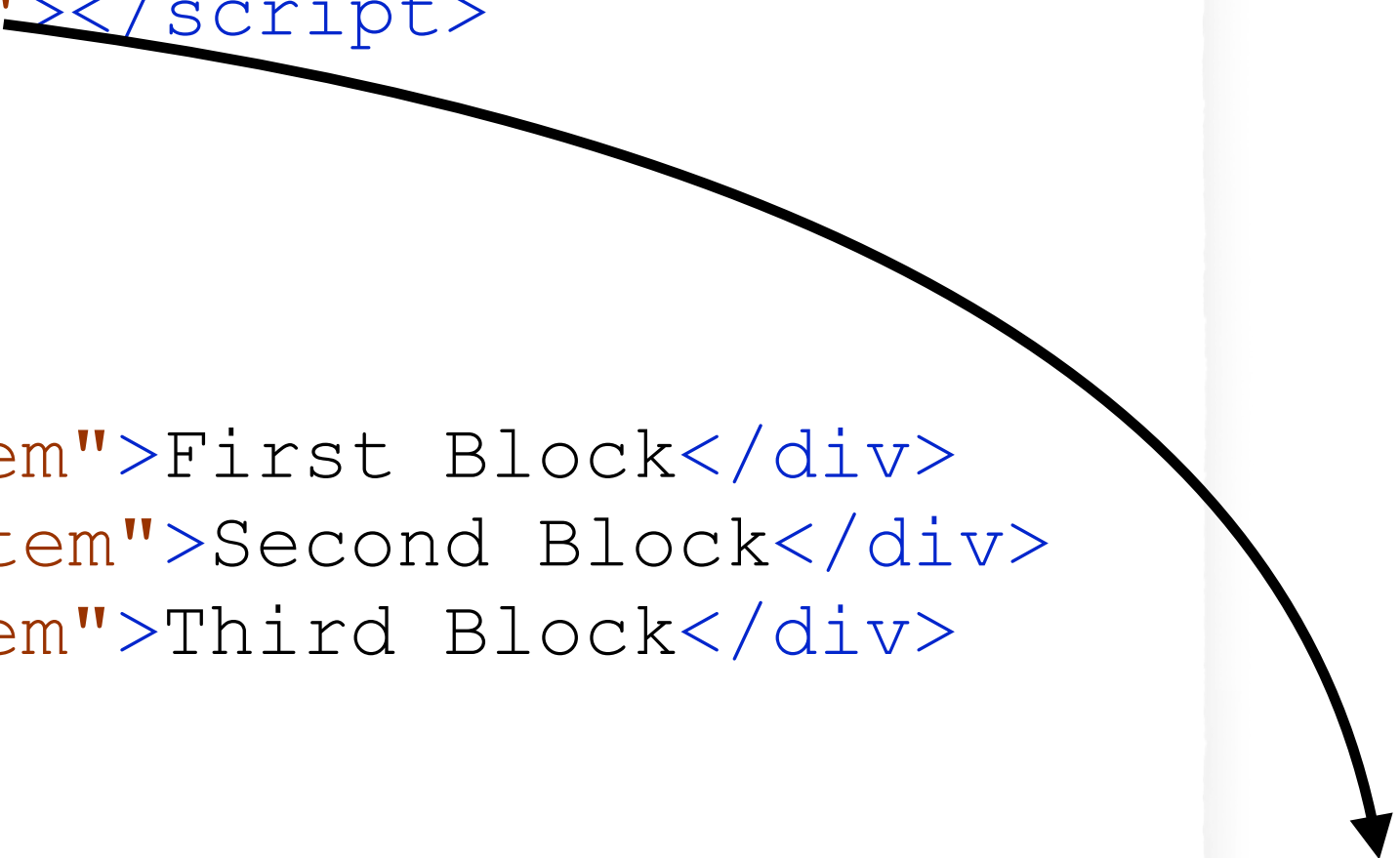
- There's also a `window` object that the DOM API provides for us.
- The Window object supports the `load` event, and we can register our own callback with this.
- The `load` event fires once the DOM has completed loading.

# window load Event

```
<!doctype html>
<head>
  <title>js/window-load.html</title>
  <link rel="stylesheet" type="text/css"
        href="getElements.css" />
  <script src="window-load.js"></script>
</head>

<body>
  <div id="main">
    <div id="first" class="item">First Block</div>
    <div id="second" class="item">Second Block</div>
    <div id="third" class="item">Third Block</div>
  </div>

</body>
</html>
```

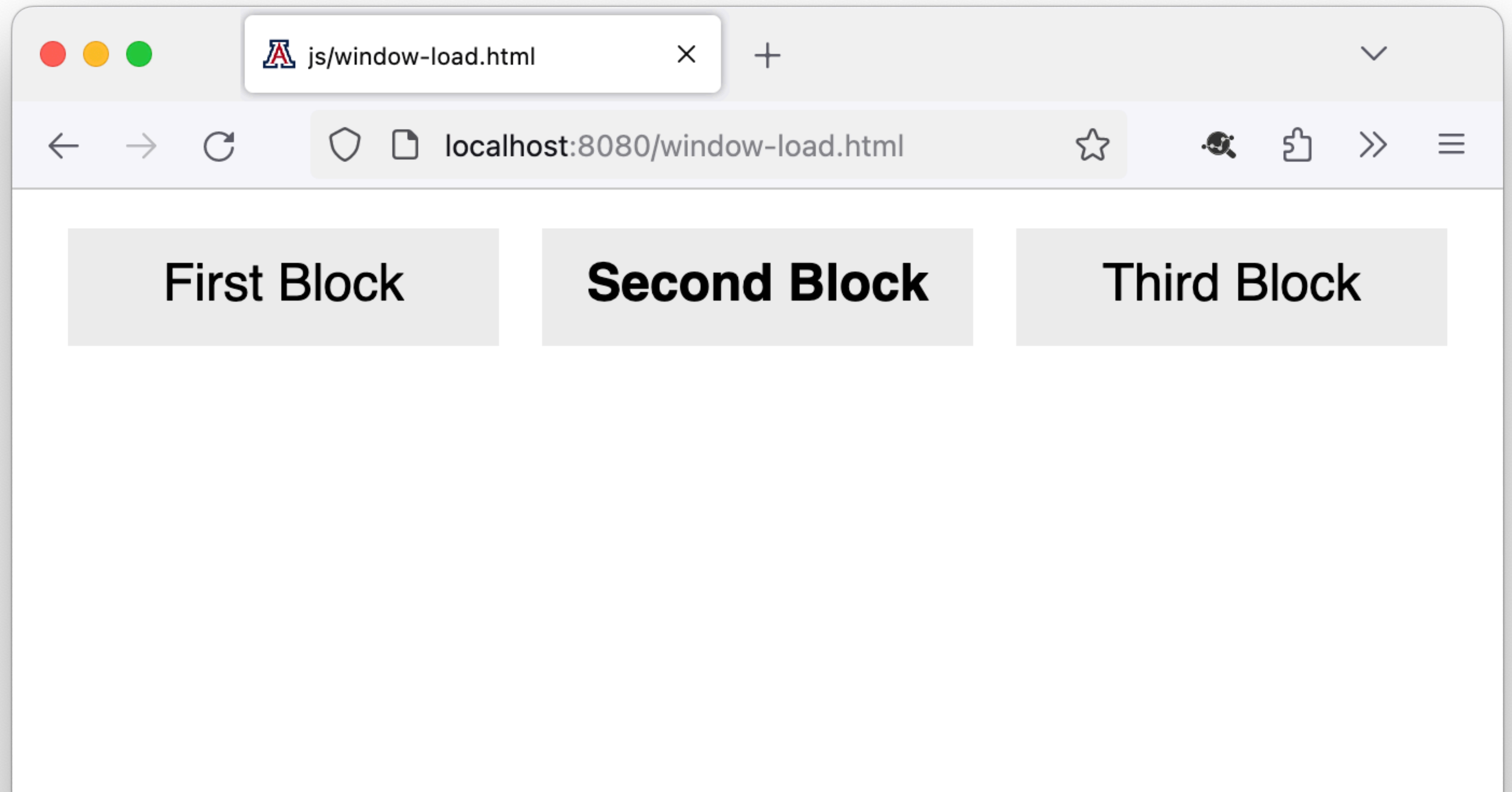


```
window.addEventListener('load', function() {
  d2 = document.getElementById('second');
  d2.classList.add('selected');
});
```



# window load Event

- Works!



# window load Event

- IE 8 supported a different method, the *object*.attachEvent method.
- Even older browsers only support a single “onload” property.

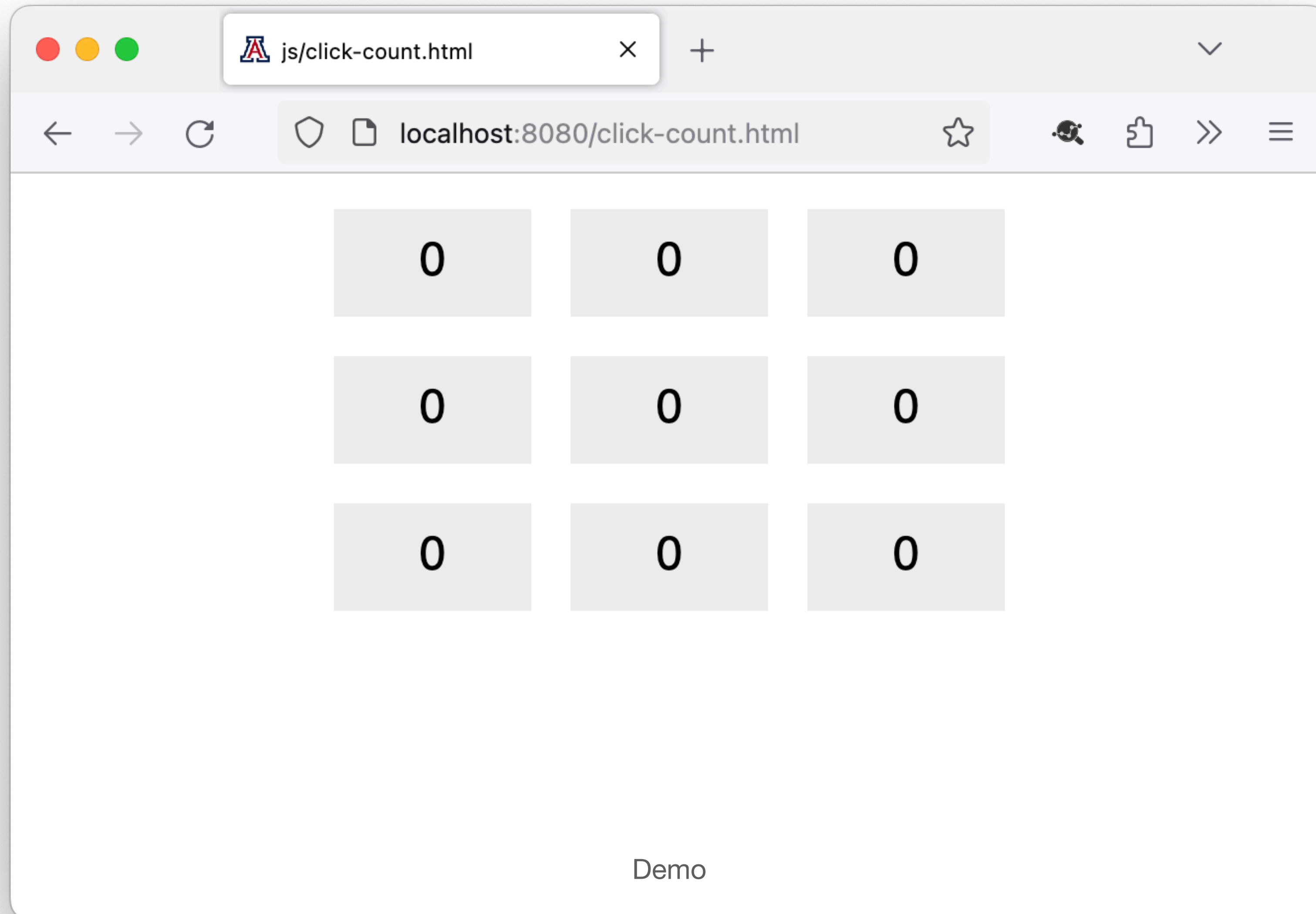
```
var ready = function(myFunciton) {
  if (window.attachEvent) {
    window.attachEvent('onload', myFunciton);
    console.log("IE");
  } else if (window.addEventListener) {
    window.addEventListener('load', myFunciton);
    console.log("Modern");
  } else {
    console.log("Legacy");
    if(window.onload) {
      var curr.onload = window.onload;
      var newonload = function() {
        curr.onload();
        myFunciton();
      };
      window.onload = newonload;
    } else {
      window.onload = myFunciton;
    }
  }
}
```

# window load Event

- IE 8 supported a different method, the *object.attachEvent* method.
- Even older browsers only support a single “onload” property.
- But none of this matters anymore because we live in the future! 🎉

```
var ready = (function(myFunciton) {  
  if (window.Event) {  
    window.addEventListener('onload', myFunciton);  
    console.log('ready');  
  } else if (window.attachEvent) {  
    window.attachEvent('onload', myFunciton);  
    console.log('ready');  
  } else {  
    console.log('ready');  
    if(window.onload) {  
      var cur = window.onload;  
      window.onload = function() {  
        cur();  
        myFunciton();  
      };  
    }  
  }  
})
```

# Putting Pieces Together



# click-count.html

```
<!doctype html>
<head>
  <title>js/click-count.html</title>
  <link rel="stylesheet" type="text/css"
        href="click-count.css"/>
  <script src="click-count.js"></script>
</head>

<body>
  <div id="main">
  </div>
</body>
</html>
```

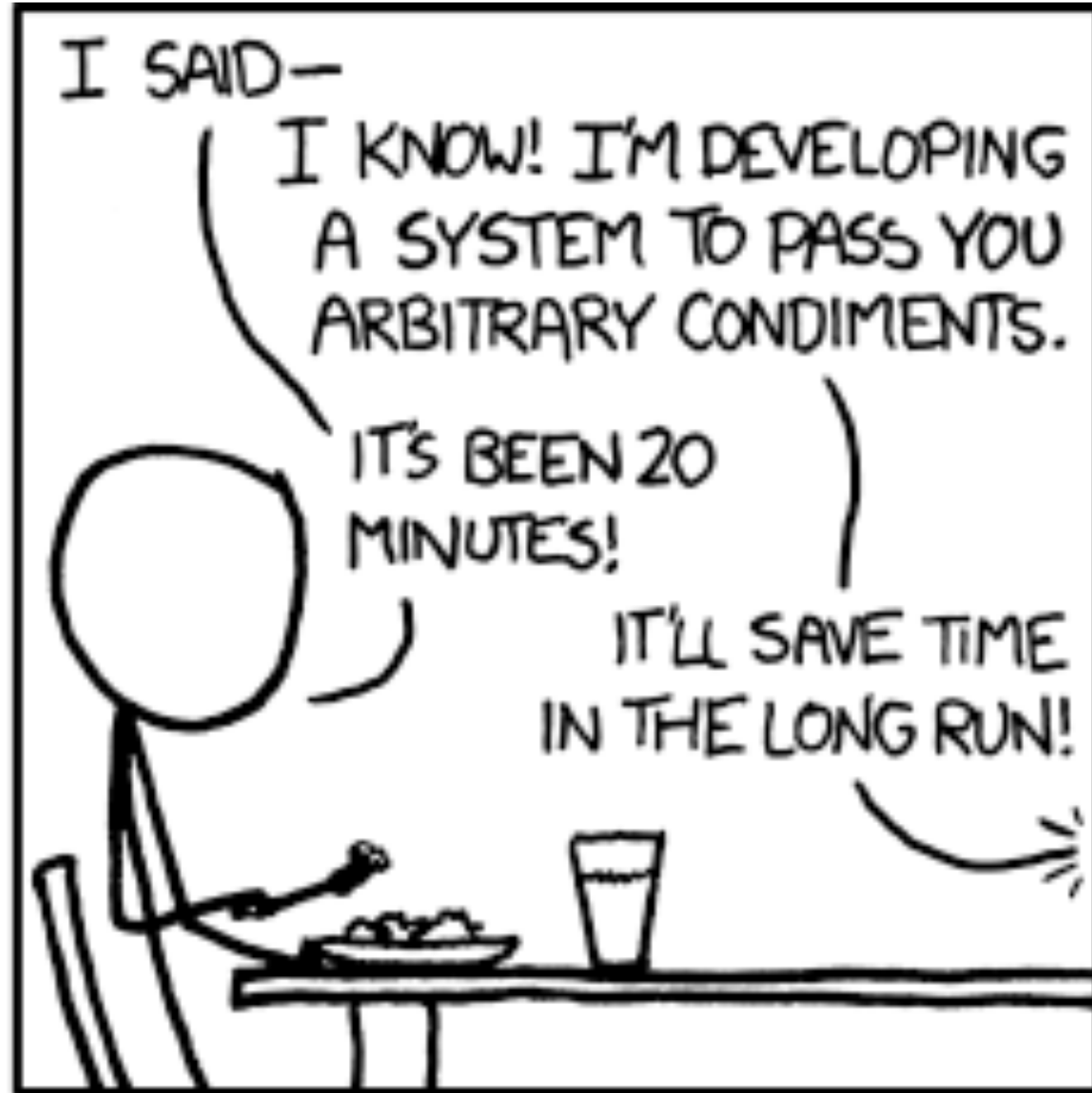
# click-count.js

```
var addCount = function(event) {
    var curCount = Number(this.textContent);
    curCount++;
    this.textContent = curCount.toString();
}

window.addEventListener('load', function() {
    var numBoxes = 9;
    main = document.getElementById('main');
    for (i = 0; i < numBoxes; i++) {
        var newBox = document.createElement("div");
        newBox.textContent = "0";
        newBox.addEventListener('click', addCount);
        main.appendChild(newBox);
    }
});
```

# Classes

**Javascript kind of has classes now?**





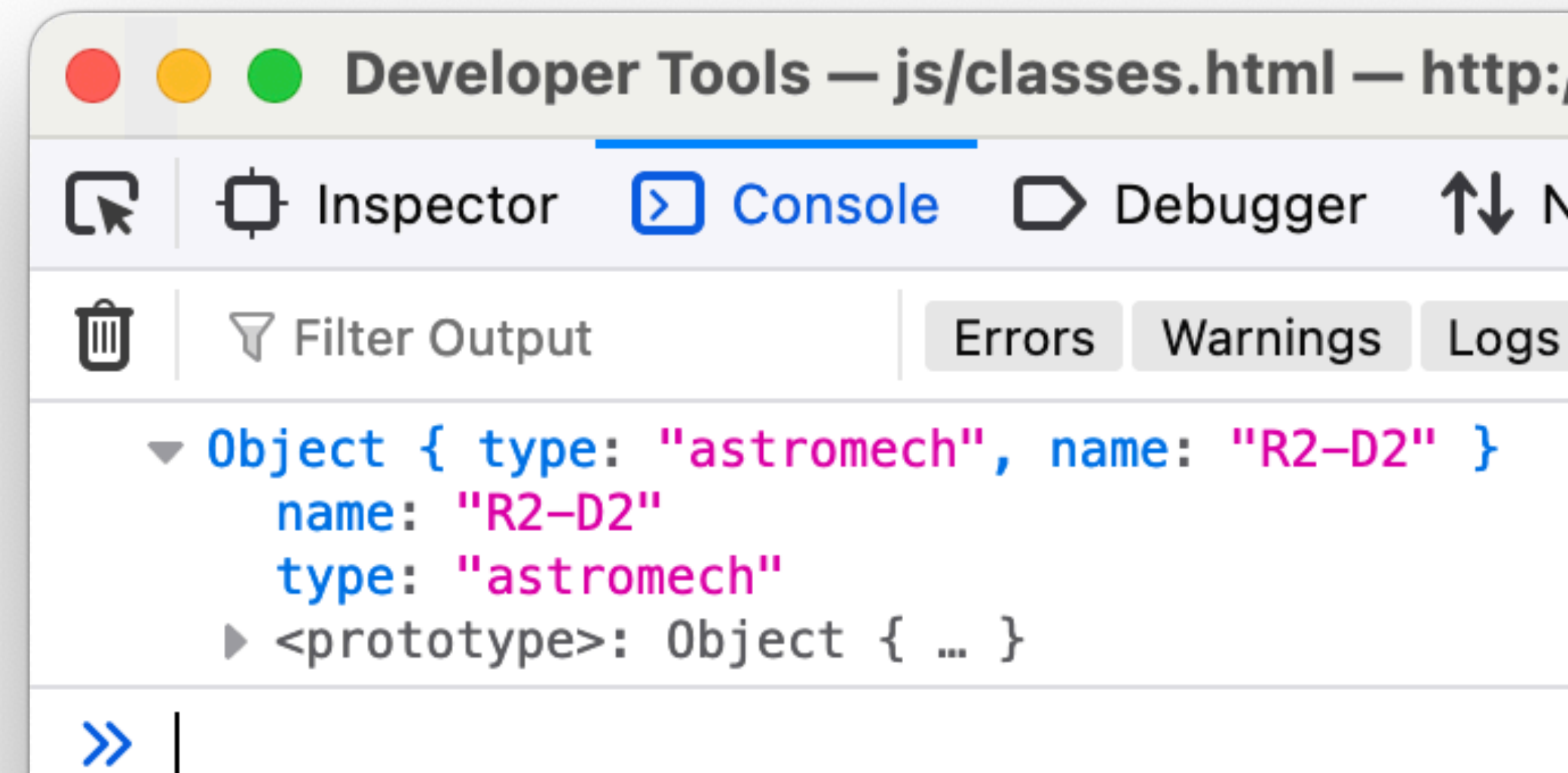
# Class Like Thingies

- Up until recently, Javascript has no “Class” concept.
- Objects are based on building on a prototype.
- “Instances” are not tied to a particular static Class definition.
- functions?

# functions and new

- Classes are just functions!
- Create new instances with the new keyword.

```
function Droid(type, name) {  
  this.type = type;  
  this.name = name;  
}  
  
var r2 = new Droid('astromech', 'R2-D2');  
var c3 = new Droid('protocol', 'C3PO');  
  
console.log(r2);
```



# prototypes

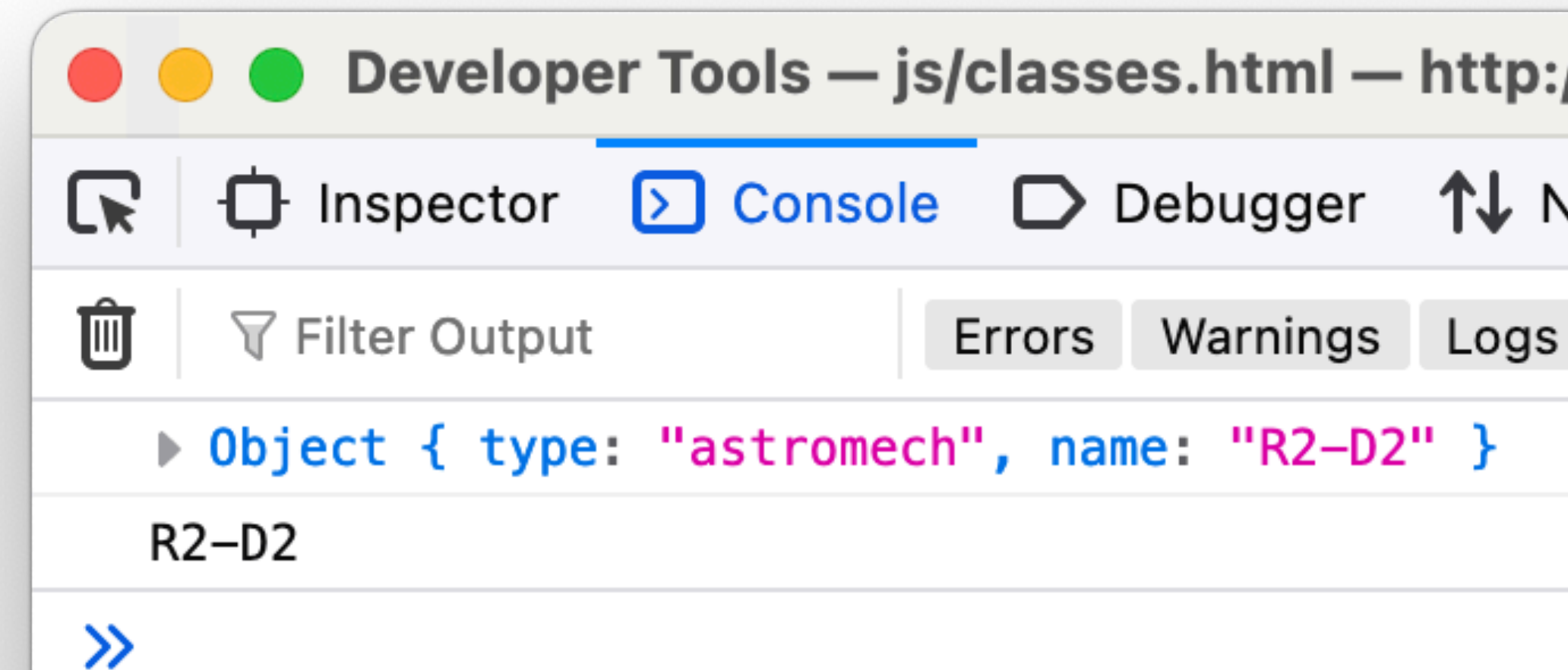
- Methods can be added through the special `.prototype` property of objects.

```
function Droid(type, name) {
  this.type = type;
  this.name = name;
}

Droid.prototype = {
  getName: function() { return this.name },
  getType: function() { return this.type }
}

var r2 = new Droid('astromech', 'R2D2');
var c3 = new Droid('protocol', 'C3PO');

console.log(r2);
console.log(r2.getName());
```



# prototypes

- Can't we convert these to Arrow Functions?
- Yes, but we can't use the `.prototype` style.
- Remember functions are first class citizens. Can be assigned directly as the value of a property.
- Arrow functions make `'this'` complicated.

```
function Droid(type, name) {  
  this.type = type;  
  this.name = name;  
  this.getName = () => this.name,  
  this.getType = () => this.type  
}  
  
var r2 = new Droid('astromech', 'R2-D2');  
var c3 = new Droid('protocol', 'C3PO');  
  
console.log(r2);  
console.log(r2.getName());
```

# prototypes

- Don't like the behavior of something? Re-define it on the fly

```
function Droid(type, name) {
  this.type = type;
  this.name = name;
}

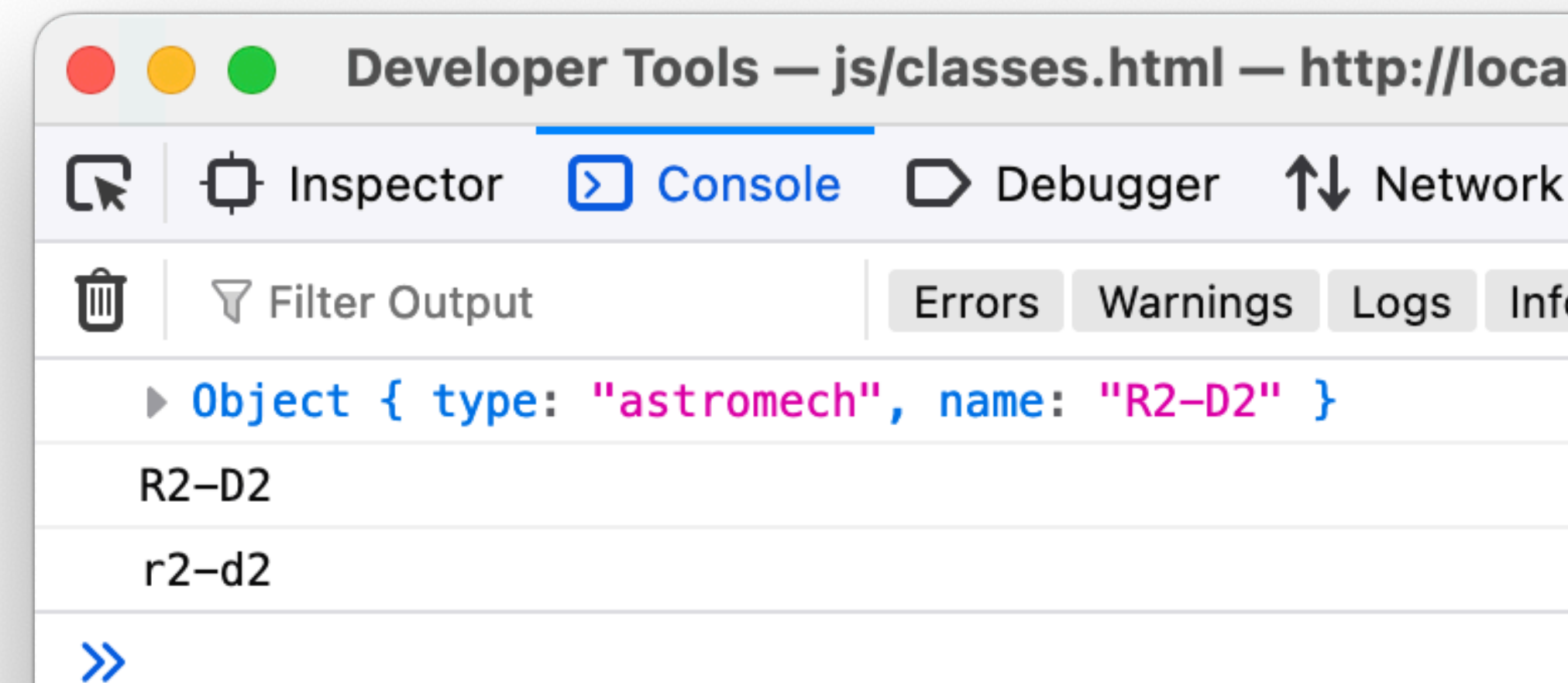
Droid.prototype = {
  getName: function() { return this.name },
  getType: function() { return this.type }
}

var r2 = new Droid('astromech', 'R2D2');
var c3 = new Droid('protocol', 'C3PO');

console.log(r2.getName());

Droid.prototype.getName =
  function() { return this.name.toLowerCase() };

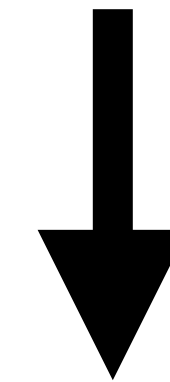
console.log(r2.getName());
```



# ES2015 Classes

- Around 2015 Javascript added some extra syntax to make class definitions a little more straight forward.
- This is mostly accomplished with preprocessor manipulation. Under the hood it is still functions and prototypes.

```
function Droid(type, name) {  
  this.type = type;  
  this.name = name;  
}  
  
Droid.prototype = {  
  getName: function() { return this.name },  
  getType: function() { return this.type }  
}
```



```
class Droid {  
  constructor(type, name) {  
    this.type = type;  
    this.name = name;  
  }  
  getName() { return this.name; }  
  getType() { return this.type; }  
}
```

# Asynchronous JavaScript

## The JavaScript Event Loop

- The JavaScript Event Loop allows for asynchronous operation
- Required for the event driven architecture
- Browser can still process events like scrolling and mouse clicks while it waits for an external network call to complete
- Two main ways to deal with events that happen over time, typically I/O operations
  - Callbacks
  - Promises

# Asynchronous JavaScript

## Callbacks

- Register a callback function to be executed when an event occurs.

```
window.addEventListener('load', function() {  
    console.log("Page has loaded")  
});
```

```
window.addEventListener('load', () =>  
    console.log("Page has loaded")  
);
```



# Asynchronous JavaScript

## Callbacks

- Register a callback function to be executed when an event occurs.
- JavaScript stores the anonymous function on the stack, and links it to the load event. When the window finishes loading, the JavaScript engine calls all registered callback functions for that event.

```
() => console.log("Page has loaded")
```

# Asynchronous JavaScript

## Callbacks

- This model works well for simple workflows, but has problems when you need to perform multiple asynchronous tasks in a specific order.
- This quickly becomes cumbersome and has come to be known as “callback hell” or “the dreaded callback pyramid”

```
window.addEventListener('load', function() {  
  callExternalAPI(apiURL, function(response) {  
    console.log(response.json())  
  })  
});
```

# Asynchronous JavaScript

## Promises

- Promises arose to combat this model, and fix many of its shortcomings
- A Promise object wraps an asynchronous operation to manage its eventual completion or failure
- Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

```
callExternalAPI (apiURL)  
  .then ( (result) => callAnotherAPI (api2URL, result) )  
  .then ( (response) => console.log (response.json () ) )
```

# Asynchronous JavaScript

## Promises

- Newer APIs such as the `fetch` API, support Promises by default, but many older ones don't. For example the `window.addEventListener` method.
- Can wrap these functions in a new Promise object
- Useful for combining new Promise code with older APIs

```
const windowEvent = (e) =>
  new Promise((resolve) => window.addEventListener(e, resolve))

windowEvent('load')
  .then(() => console.log("Page Load Promise Resolved"))
```

# **AJAX**

**Talking Behind the Browser's Back**

# AJAX

- **A**synchronous
- **J**avascript
- **A**nd
- **X**ML

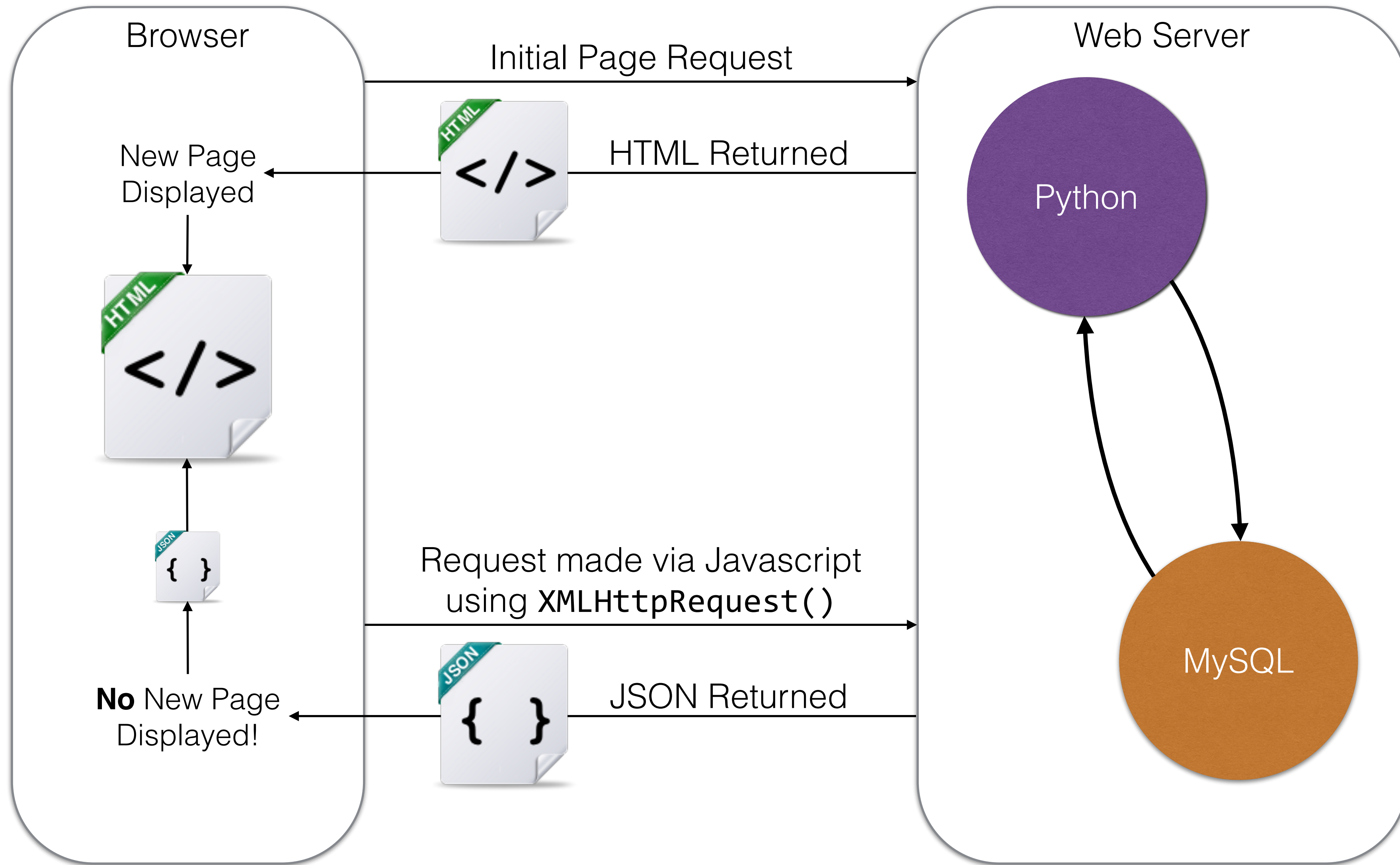
# AJAX

- Fortunately, almost no one uses XML anymore
- JSON is far more popular now as the format for asynchronous data
- **JavaScript Object Notation**

# XMLHttpRequest

- Concept First proposed by Microsoft
- First appeared in IE 5 as an ActiveX component
- Mozilla Adopted the idea and created a Javascript implementation of it as **nsIXMLHttpRequest**. Appeared in Gecko engine in 2002
- Became the *de facto* standard when WebKit implemented it in 2004
- W3C formally standardized it in 2006





# **XMLHttpRequest Demo**

**<https://www2.cs.arizona.edu/classes/cs346/spring24/docs/examples/ajax-demo/>**

# New fetch API

## So much simpler

- We can finally put together all our function and Promise knowledge

```
fetch(apiURL)
  .then((responsePromise) => responsePromise.json())
  .then((responseObj) => process(responseObj))
```

**Now we know enough to be  
dangerous**