

Javascript

Because ECMAScript sounds horrible

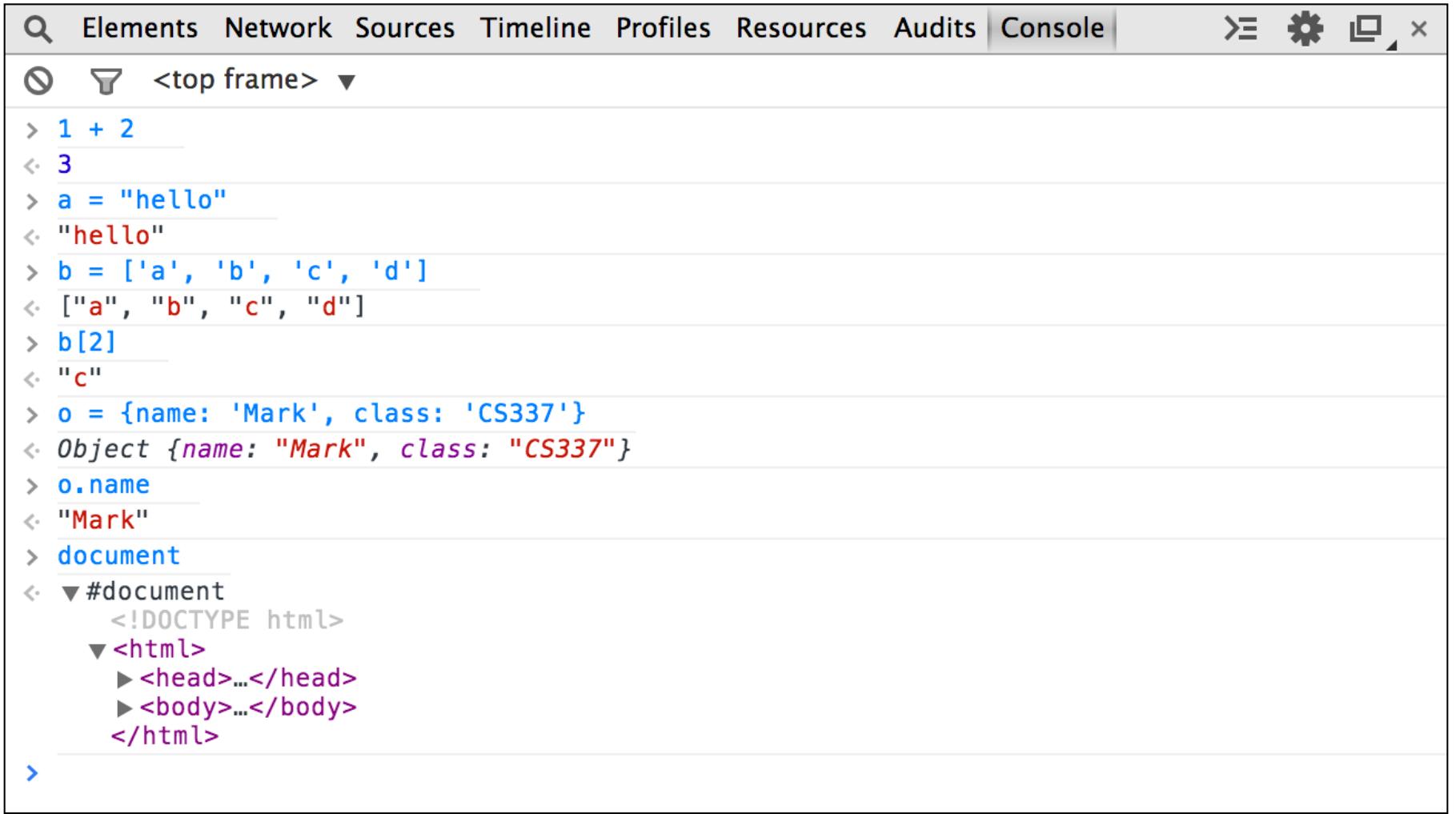
Javascript

- 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's developer tools interface with the "Console" tab selected. The console window displays a series of Javascript commands and their results, along with a hierarchical view of the current document's DOM structure.

```
<top frame>
> 1 + 2
<- 3
> a = "hello"
<- "hello"
> b = ['a', 'b', 'c', 'd']
<- ["a", "b", "c", "d"]
> b[2]
<- "c"
> o = {name: 'Mark', class: 'CS337'}
<- Object {name: "Mark", class: "CS337"}
> o.name
<- "Mark"
> document
<- #document
  <!DOCTYPE html>
  <html>
    <head>...</head>
    <body>...</body>
  </html>
>
```

Documentation

http://ecma262-5.com/ELS5_HTML.htm

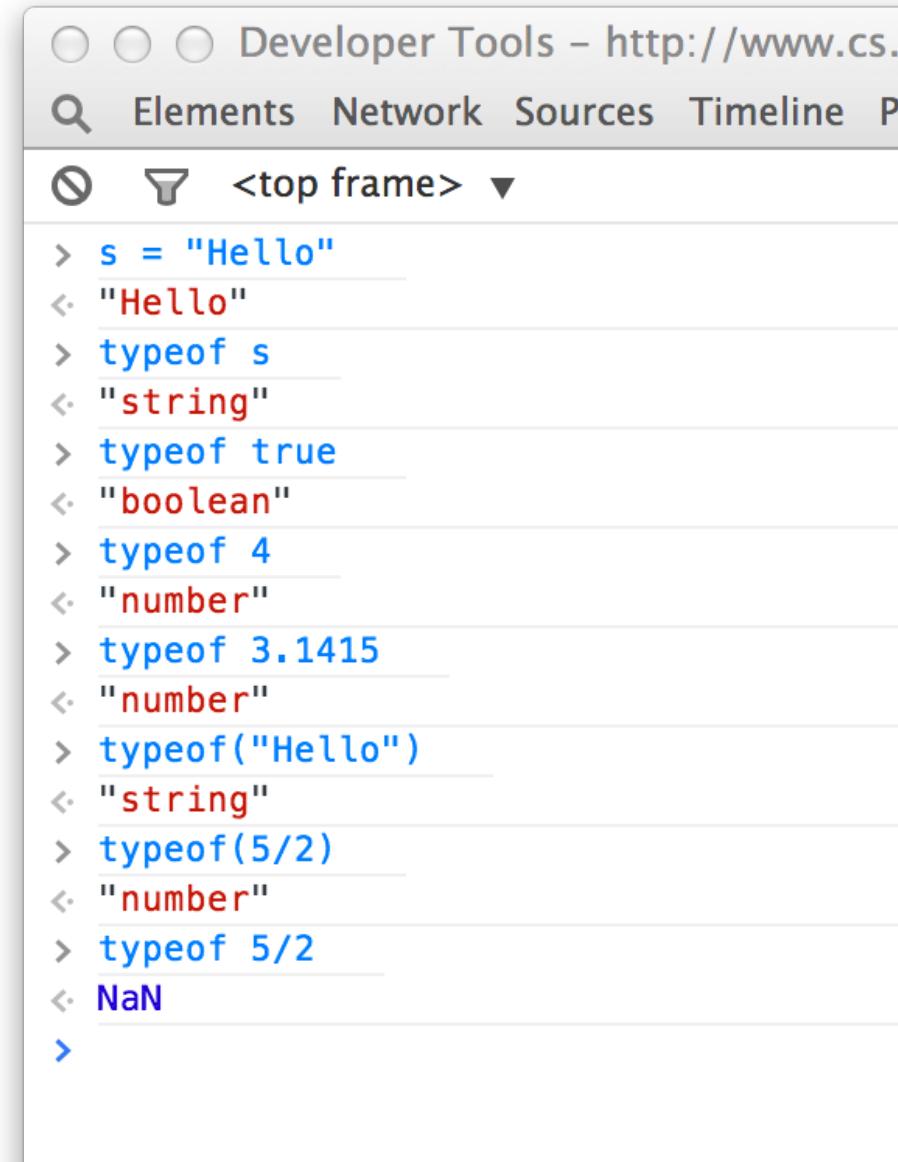
<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference>

Data Types

- Basic Data Types
 - number
 - boolean
 - string
 - object

Data Types

- `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`

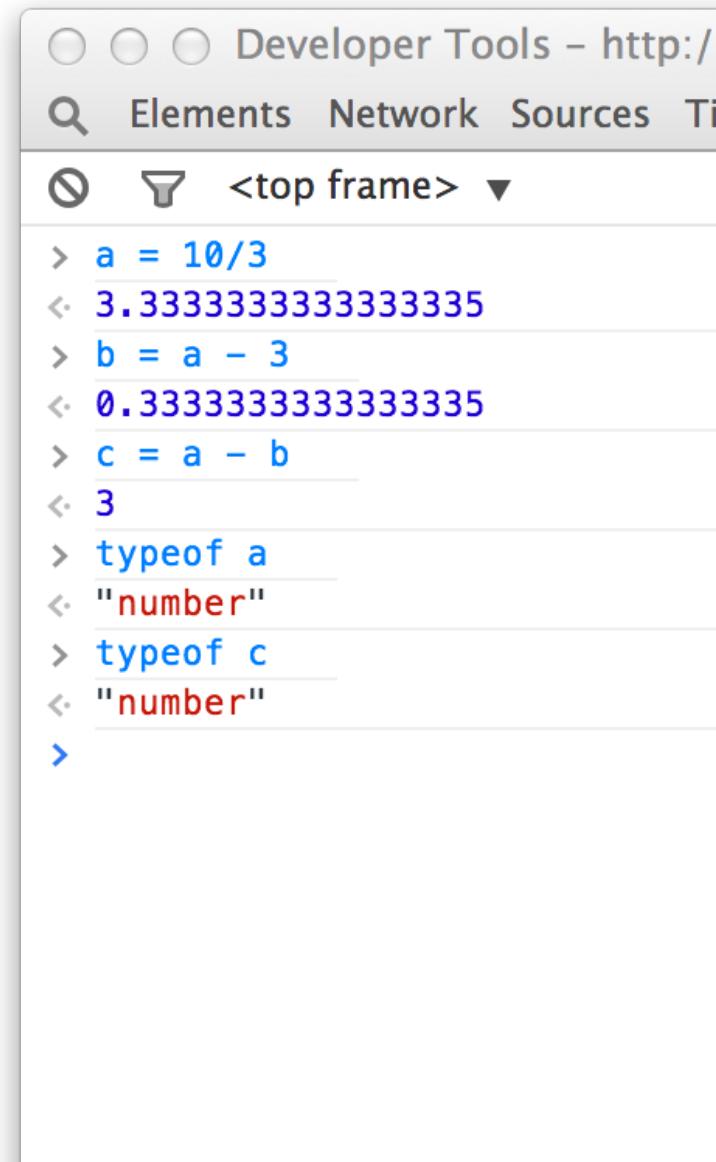


The screenshot shows a browser's developer tools console with the title "Developer Tools – http://www.cs...". The console interface includes tabs for Elements, Network, Sources, Timeline, and Performance. Below the tabs, there is a search bar and a dropdown menu set to "<top frame>". The main area displays a list of typed commands and their results:

- > `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.



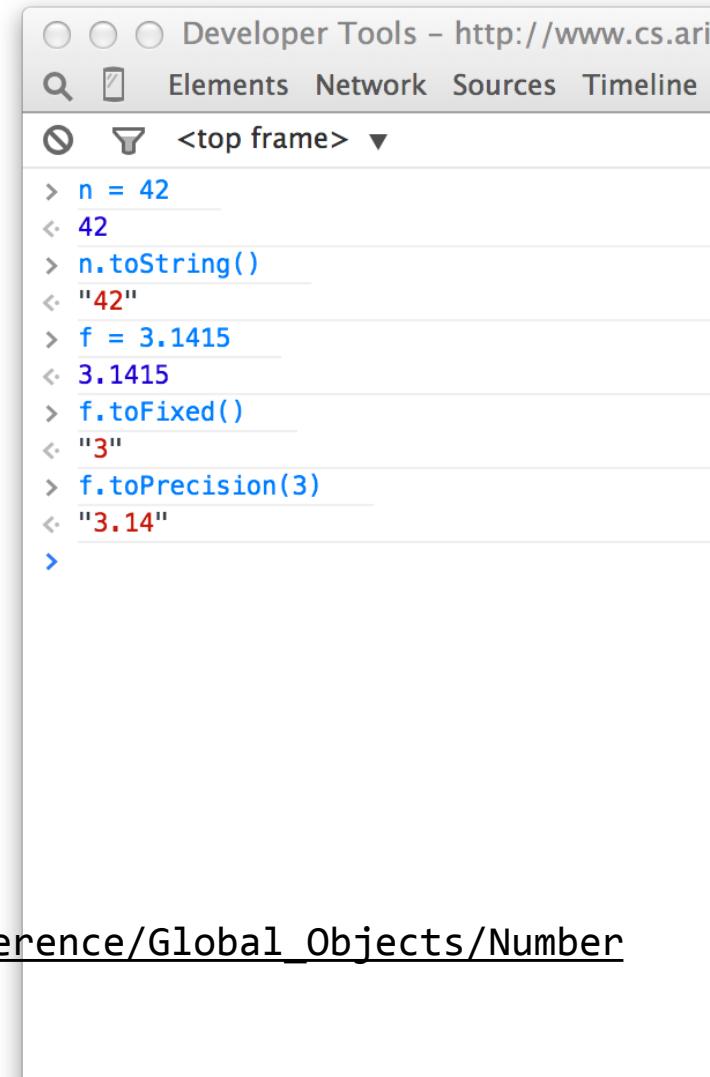
The screenshot shows a developer tools window with the title "Developer Tools - http://". The "Console" tab is active, displaying the following command history:

```
<top frame>
> a = 10/3
< 3.3333333333333335
> b = a - 3
< 0.3333333333333335
> c = a - b
< 3
> typeof a
< "number"
> typeof c
< "number"
```

The output values for 'a' and 'c' are shown in blue, while the types ('number') are shown in red. The command 'b' is also present in the history.

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 window titled "Developer Tools - http://www.cs.ariel.com". The console interface includes tabs for Elements, Network, Sources, and Timeline. The main area displays a session transcript:

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

Strings

Chrome

The screenshot shows the Chrome Developer Tools console. It displays a series of JavaScript code snippets and their results. The code includes setting variables `a` and `r` to strings containing regular characters, and then setting them to strings containing the Apple emoji character (U+1F34D). The results show the original strings and then the strings with the emoji.

```
<top frame>
> a = "is for Apple";
<- "is for Apple"
> r = "José Nuñiez";
<- "José Nuñiez"
> a = "\u267d an apple!";
<- "\u267d an apple!"
>
```

- A series of zero or more characters.
- Unicode support is pretty good.
- Browser support for full unicode support is spotty.

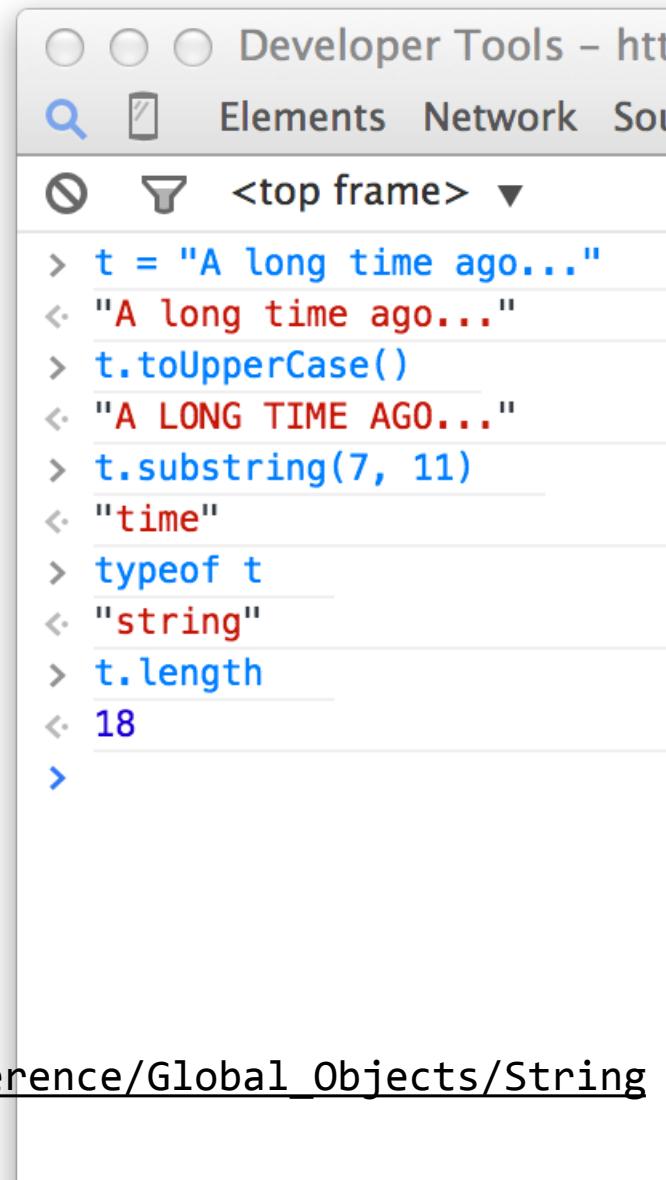
Safari

The screenshot shows the Safari Developer Tools console. It displays a similar set of JavaScript code snippets and their results to the Chrome example, demonstrating how the emoji character is rendered. The emoji character appears as a red apple icon with a bite taken out of it.

```
Console
> a = "\u267d an apple!";
<- "\u267d an apple!"
>
```

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

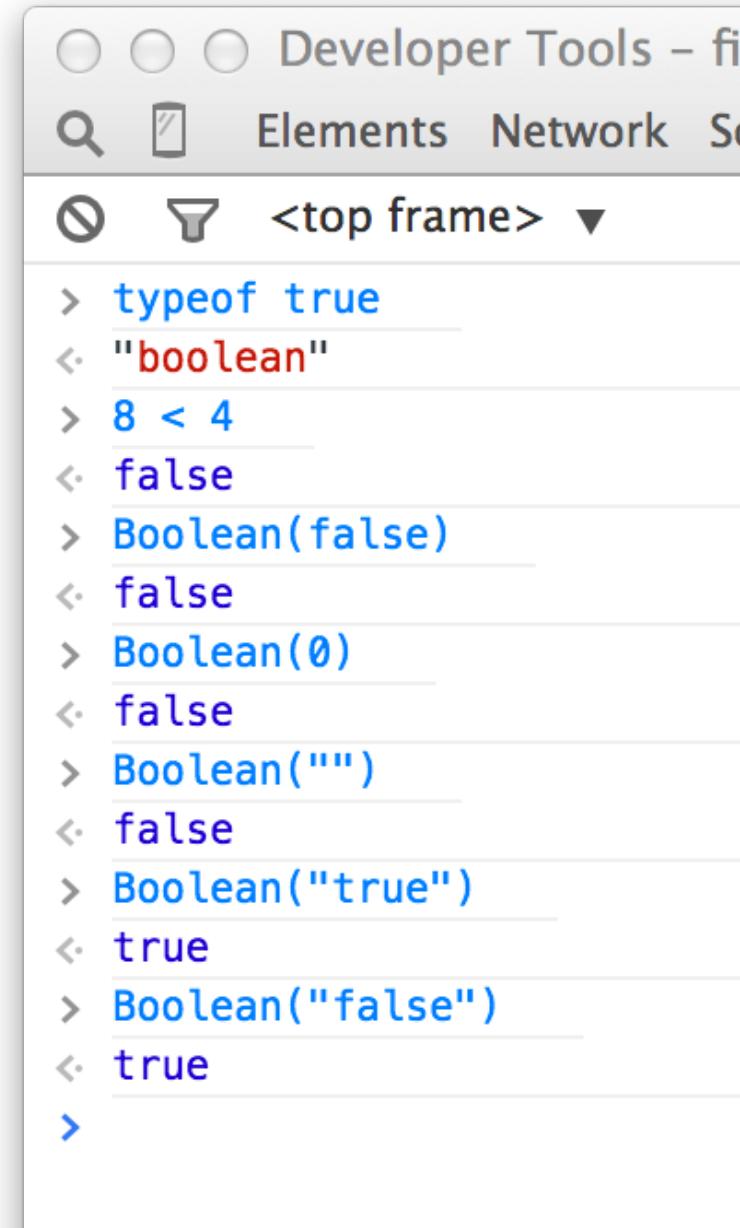


A screenshot of a browser's developer tools console window titled "Developer Tools - http://...". The console interface includes tabs for Elements, Network, and Source, along with search and filter controls. The main area displays a series of JavaScript commands and their results:

```
<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`”!



The screenshot shows a browser's developer tools console interface. At the top, there are three circular status indicators followed by the text "Developer Tools - fi". Below the toolbar, there are icons for search, refresh, and navigation, followed by the text "Elements Network Se". The main area of the console displays a series of input and output lines. The inputs are in blue, and the outputs are in red. The session starts with `> typeof true`, which outputs "`"boolean"`". It then shows `> 8 < 4` and `<- false`. The next few lines involve the `Boolean` constructor: `> Boolean(false)` outputs `false`, `<- Boolean(0)` also outputs `false`, and `<- Boolean("")` also outputs `false`. Following this, `> Boolean("true")` outputs `true`, and `<- Boolean("false")` also outputs `true`. The session concludes with a final input line starting with `>`.

```
> typeof true
<- "boolean"
> 8 < 4
<- false
> Boolean(false)
<- false
> Boolean(0)
<- false
> Boolean("")
<- false
> Boolean("true")
<- true
> Boolean("false")
<- true
>
```

Variables

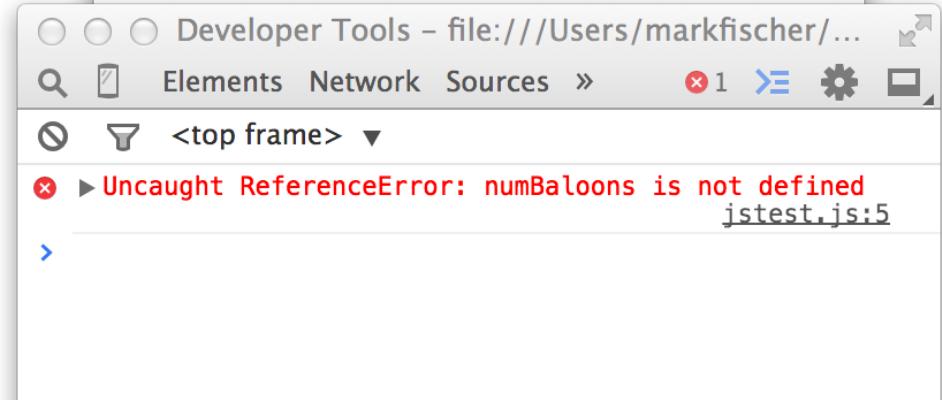
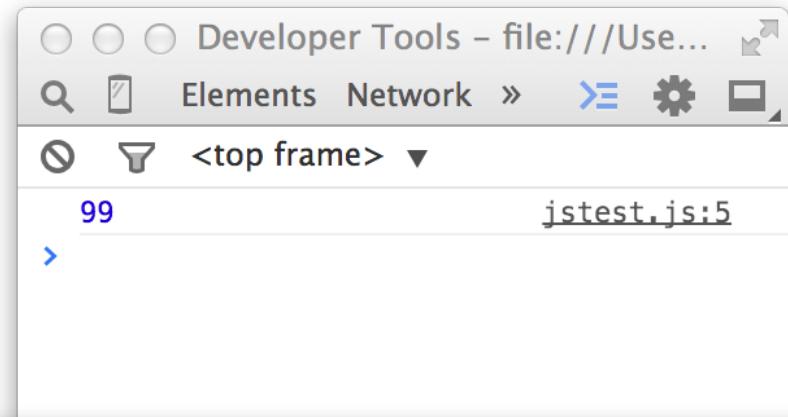
- Variable names can be any combination of letters, numbers, an underscore (_), or \$
- Variable names cannot start with a number.
- Variables do not need to be declared.
- The **var** keyword can be used to declare and scope variables.

Variables

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

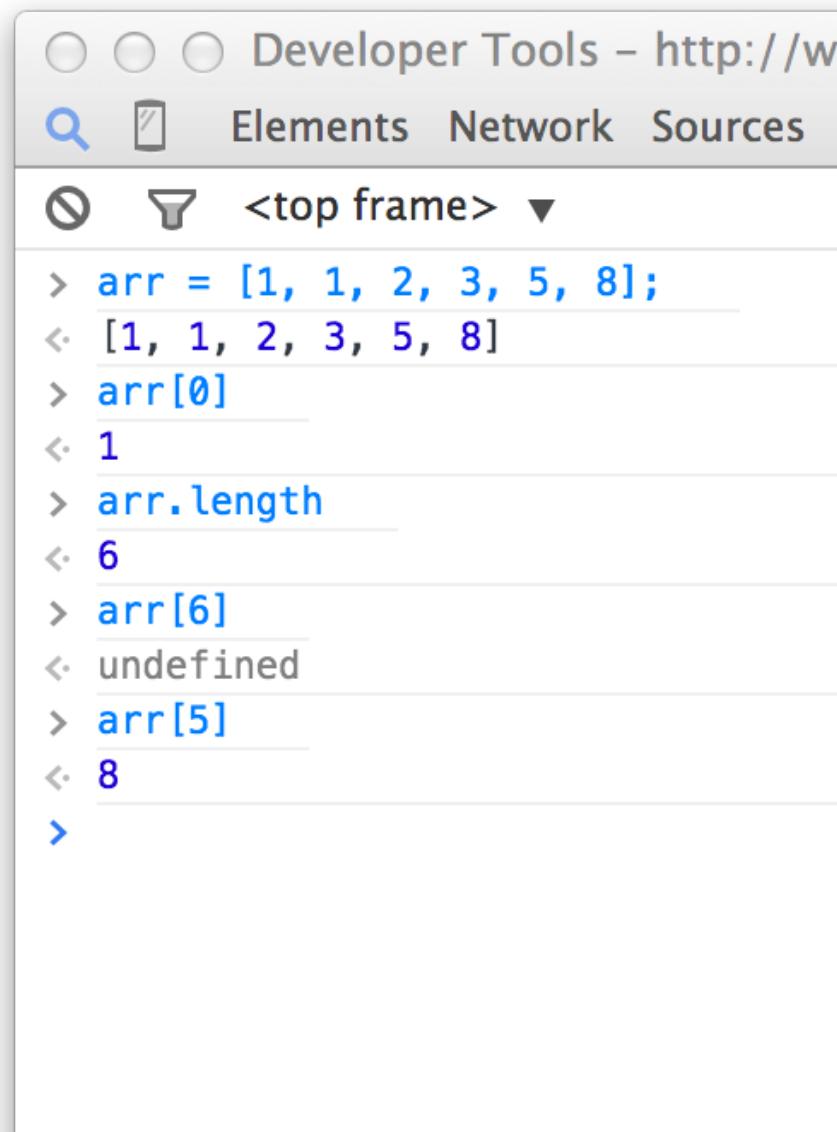
```
var foo = function() {  
    numBalloons = 99;  
}  
foo();  
console.log(numBalloons);
```

```
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

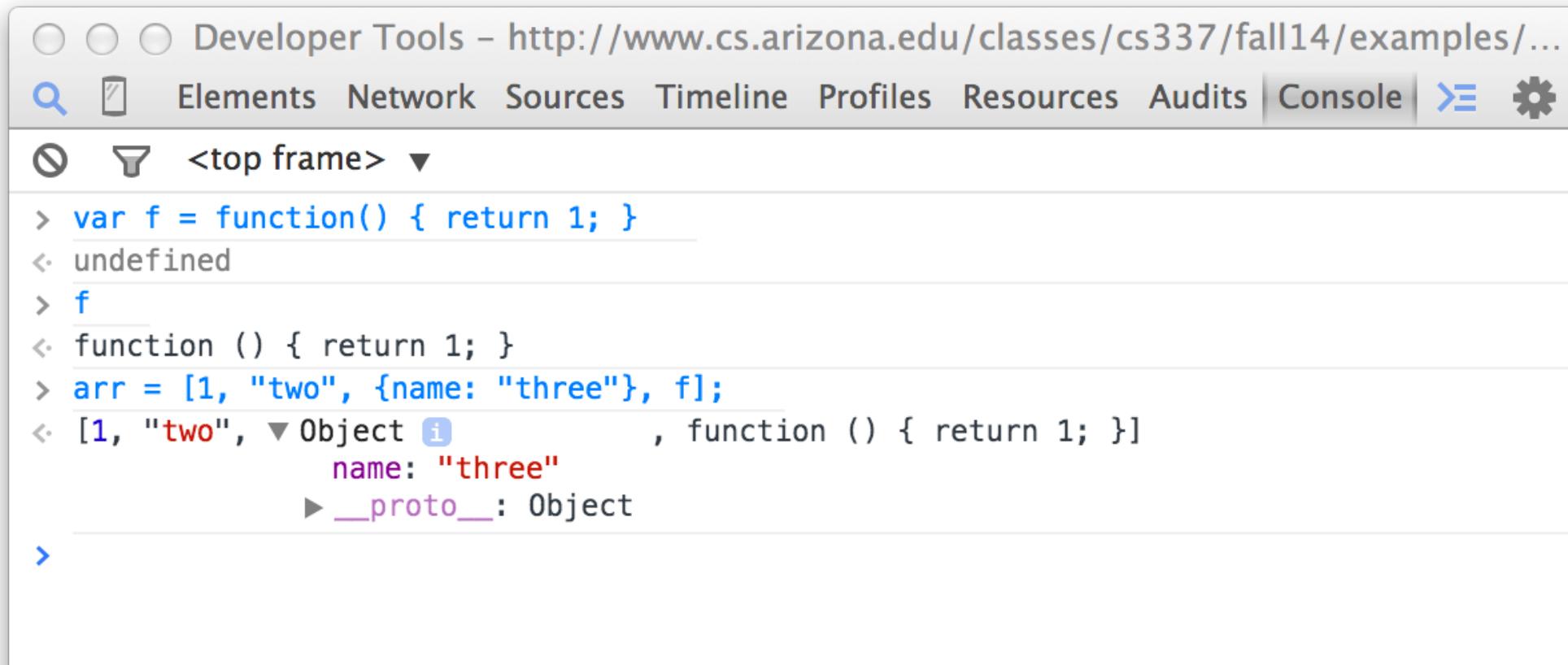


A screenshot of a browser's developer tools console window titled "Developer Tools - http://w...". The console interface includes tabs for Elements, Network, and Sources, and a search bar with a filter icon. The main area shows a series of JavaScript commands and their results:

```
<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.

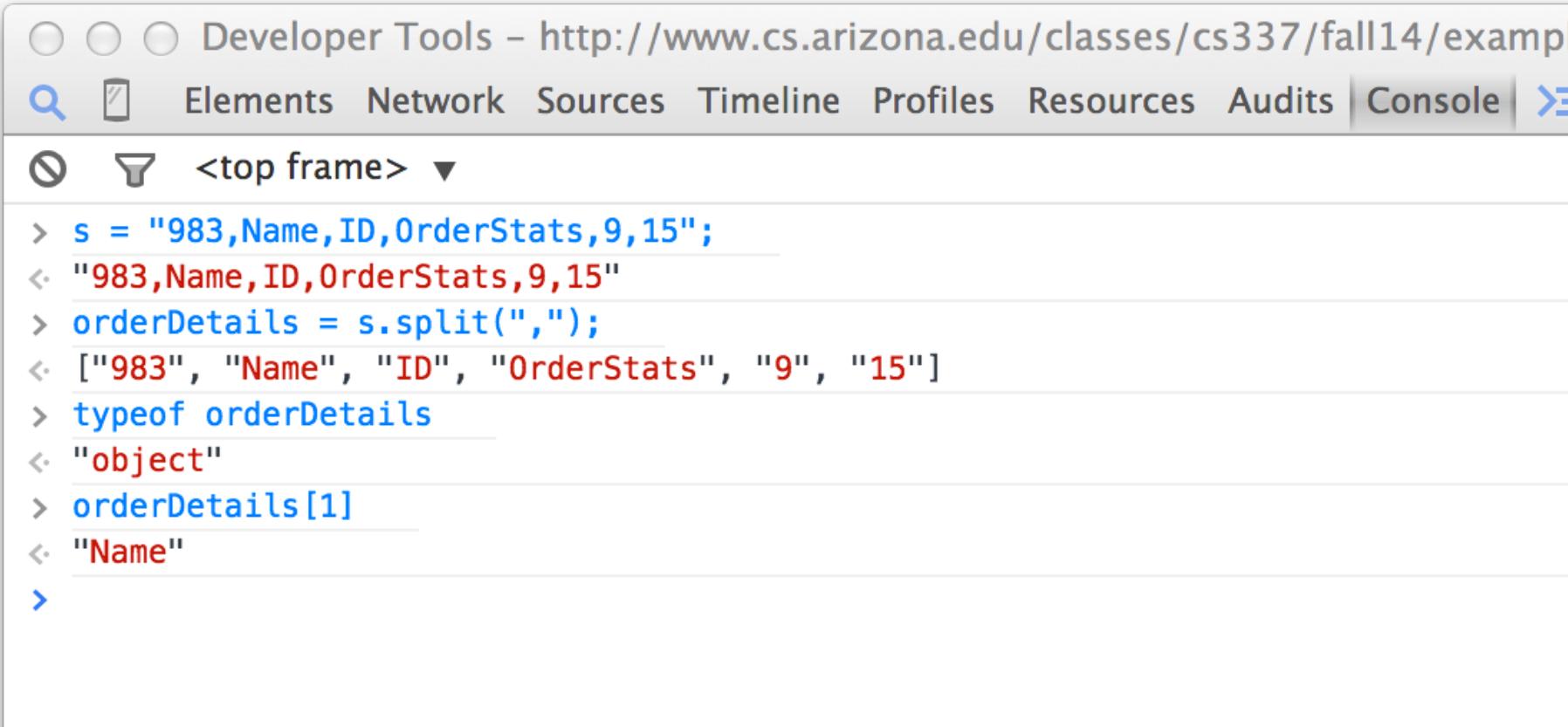


The screenshot shows the Google Chrome Developer Tools interface, specifically the Console tab. The title bar reads "Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examples/...". The menu bar includes "Elements", "Network", "Sources", "Timeline", "Profiles", "Resources", "Audits", "Console", and settings icons. The main area displays the following JavaScript session:

```
> var f = function() { return 1; }
<- undefined
> f
<- function () { return 1; }
> arr = [1, "two", {name: "three"}, f];
<- [1, "two", ▶ Object { name: "three" }, function () { return 1; }]
          name: "three"
          ► __proto__: Object
>
```

Arrays From Strings

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



The screenshot shows a browser's developer tools console window. The title bar reads "Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examp...". The menu bar includes "Elements", "Network", "Sources", "Timeline", "Profiles", "Resources", "Audits", "Console", and "Console". The main area shows the following JavaScript interaction:

```
<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"
>
```

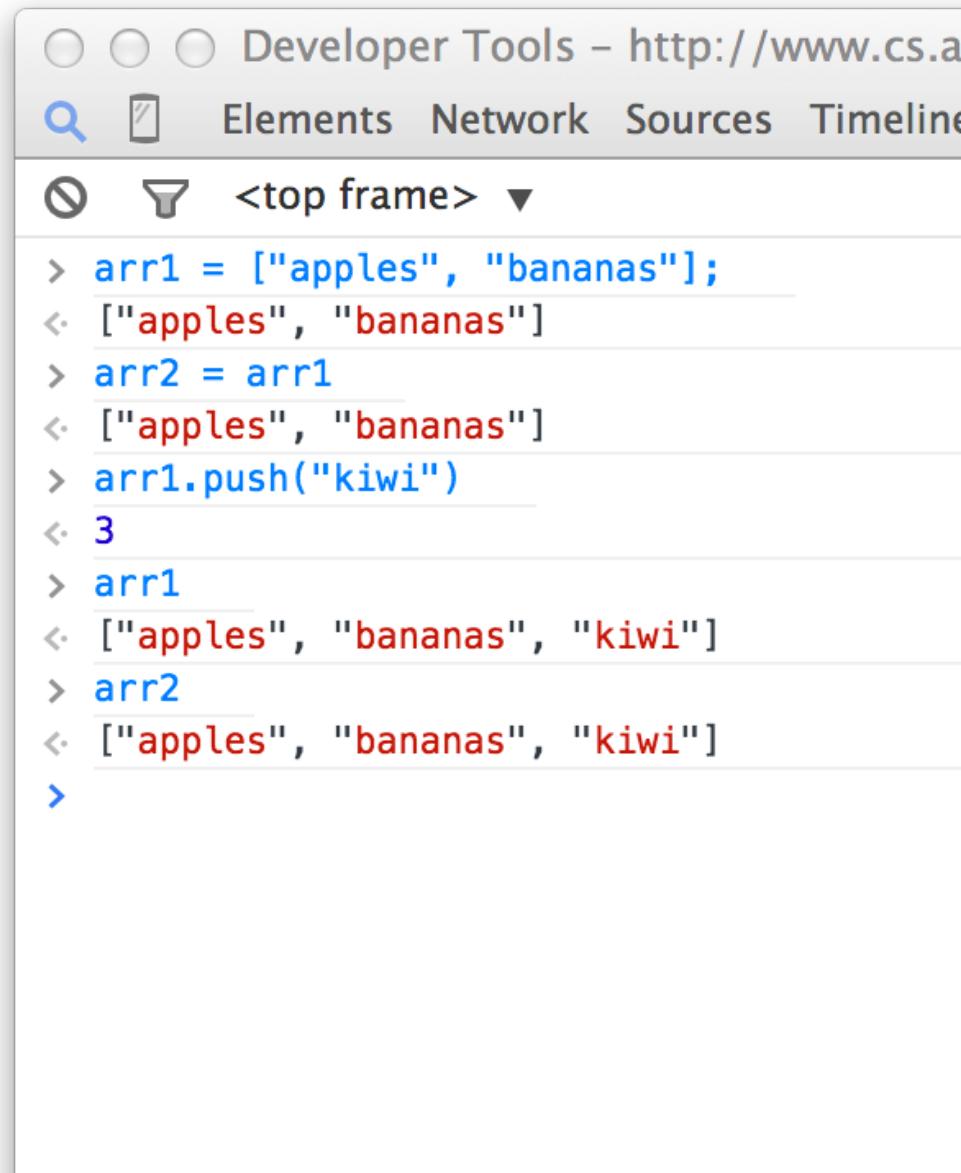
Array Methods

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

- 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.



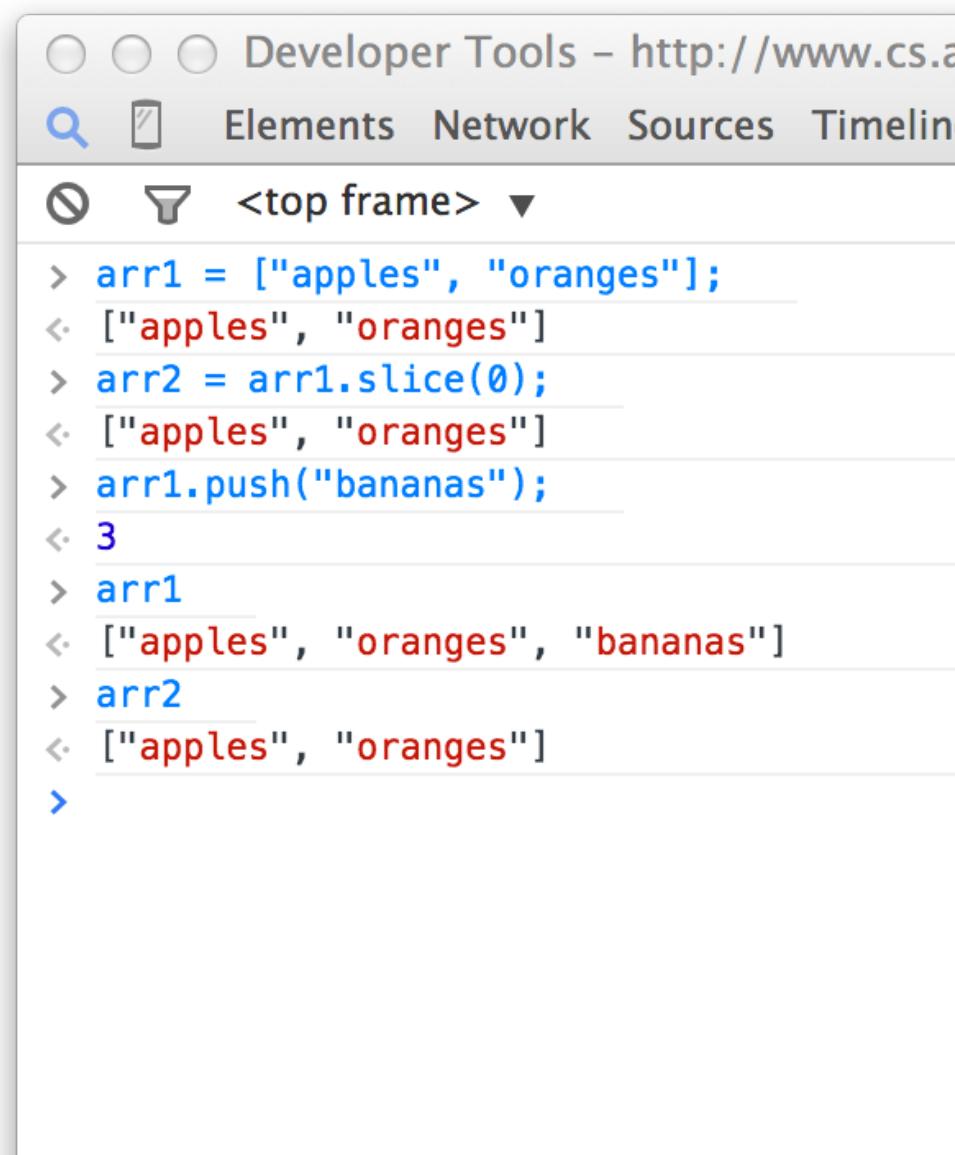
The screenshot shows a browser's developer tools console with the title "Developer Tools - http://www.cs.a...". The console interface includes tabs for Elements, Network, Sources, and Timeline, along with search and filter icons.

```
<top frame>
> arr1 = ["apples", "bananas"];
<- ["apples", "bananas"]
> arr2 = arr1
<- ["apples", "bananas"]
> arr1.push("kiwi")
<- 3
> arr1
<- ["apples", "bananas", "kiwi"]
> arr2
<- ["apples", "bananas", "kiwi"]
>
```

The code demonstrates that `arr2` is a reference to the same array as `arr1`. When `arr1` is modified (pushing "kiwi"), `arr2` also reflects this change, showing the updated array `["apples", "bananas", "kiwi"]`.

Array Assignment

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



The screenshot shows a browser's developer tools console window titled "Developer Tools – http://www.cs.a...". The tabs at the top are "Elements", "Network", "Sources", and "Timeline". The console area has a search bar and a filter icon. It displays the following code and its execution results:

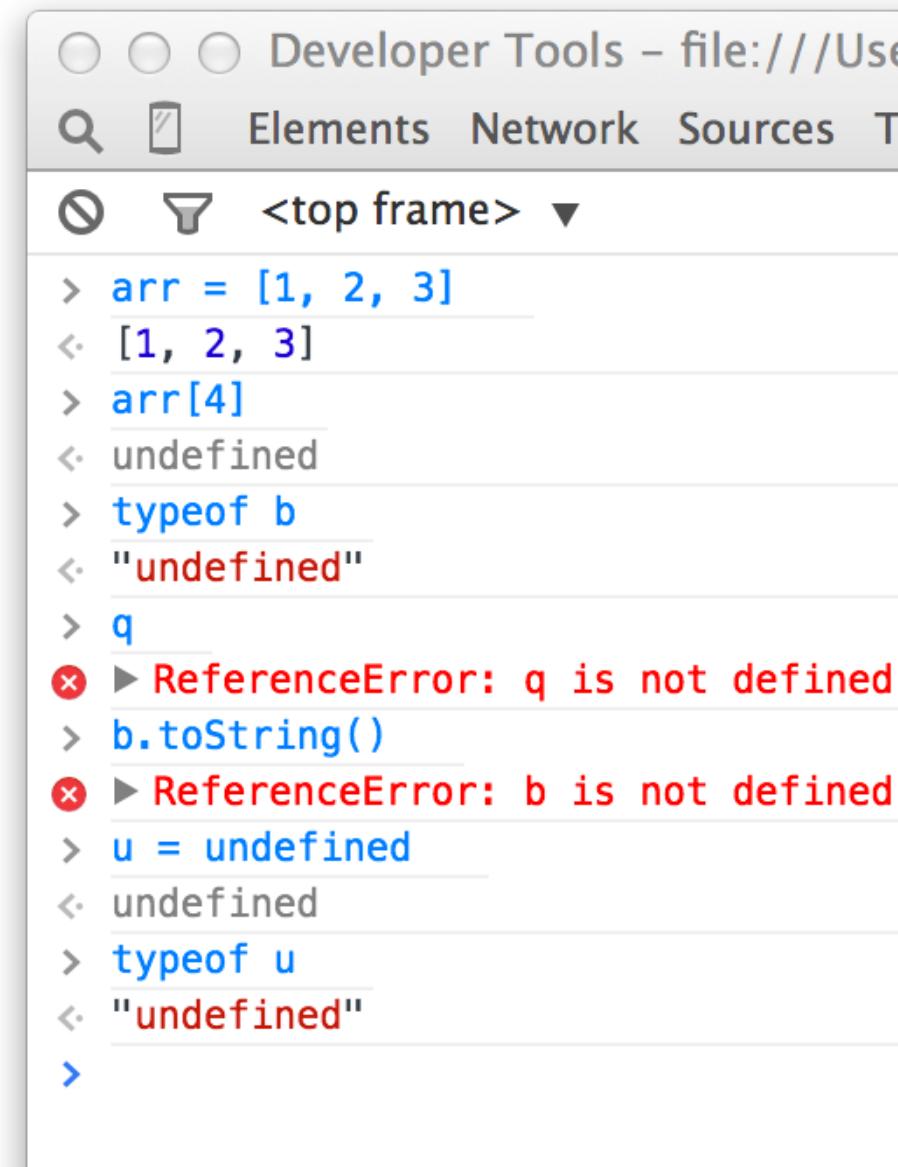
```
> arr1 = ["apples", "oranges"];
<- ["apples", "oranges"]
> arr2 = arr1.slice(0);
<- ["apples", "oranges"]
> arr1.push("bananas");
<- 3
> arr1
<- ["apples", "oranges", "bananas"]
> arr2
<- ["apples", "oranges"]
>
```

The code creates an array `arr1` with two elements: "apples" and "oranges". It then creates a new array `arr2` by copying all elements from `arr1` using the `.slice(0)` method. After this, it adds a new element "bananas" to `arr1` using the `.push` method, changing its length to 3. Finally, it prints the values of `arr1`, `arr2`, and a final empty line.

undefined

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.



A screenshot of a browser's developer tools console. The title bar says "Developer Tools - file:///User". The tabs are Elements, Network, Sources, and Timeline. Below the tabs, it says "<top frame> ▾". The console shows the following interactions:

```
> 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:

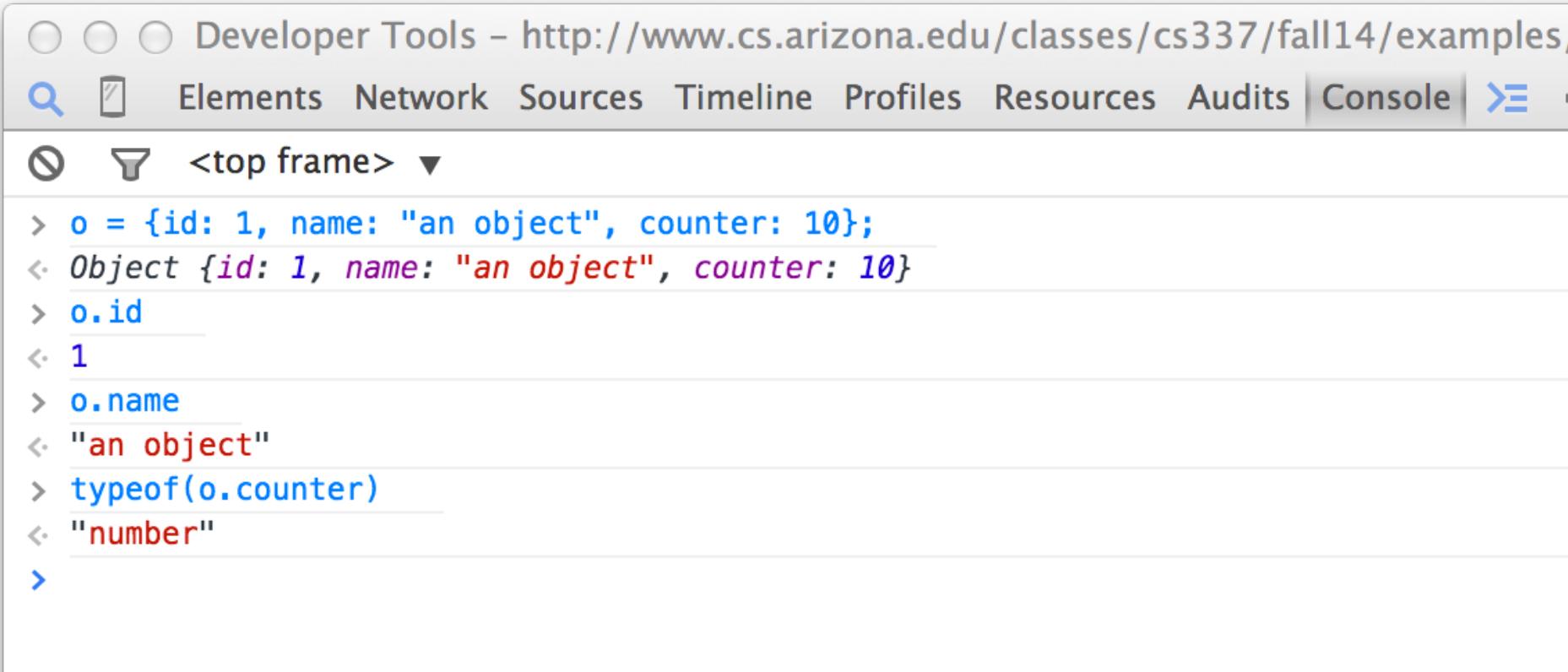
```
o = {id: 1, name: "an object", counter: 10};
```

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

Objects

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

- Access properties via dot syntax



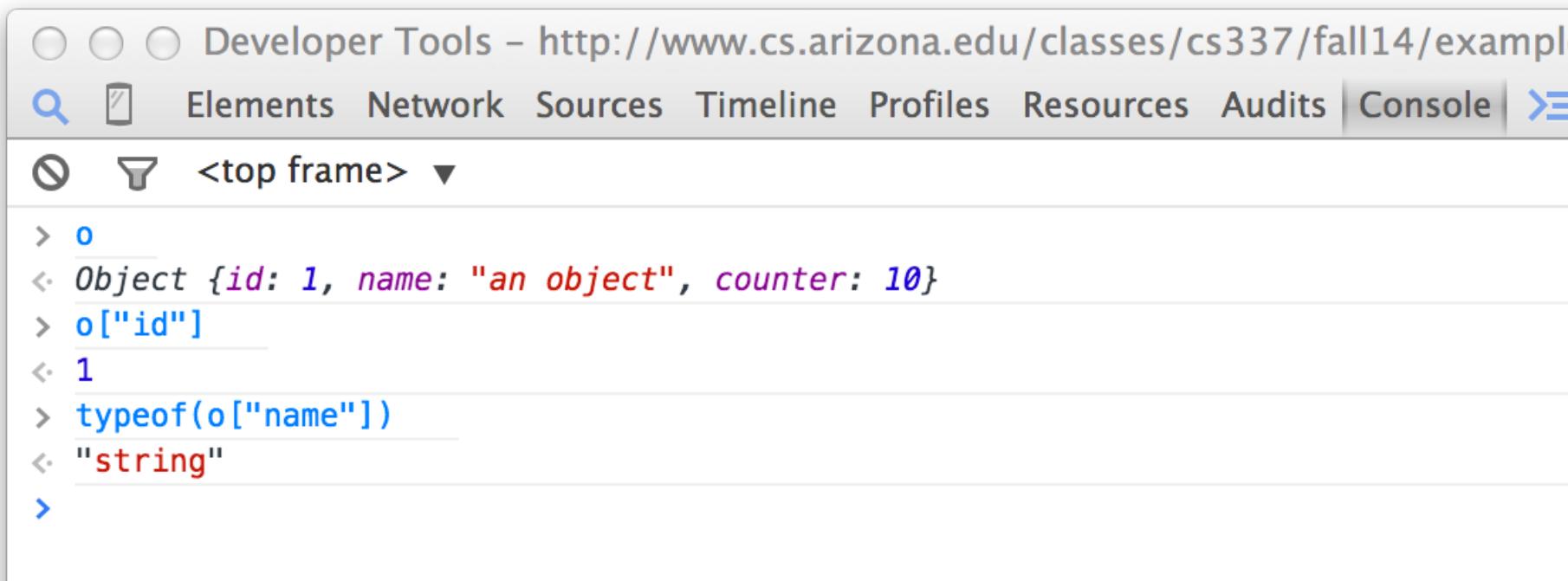
The screenshot shows a browser's developer tools console window. The title bar says "Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/examples". The tabs at the top are Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console. The Console tab is active. The console output is as follows:

```
<top frame> ▾
> o = {id: 1, name: "an object", counter: 10};
< Object {id: 1, name: "an object", counter: 10}
> o.id
< 1
> o.name
< "an object"
> typeof(o.counter)
< "number"
>
```

Objects

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

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



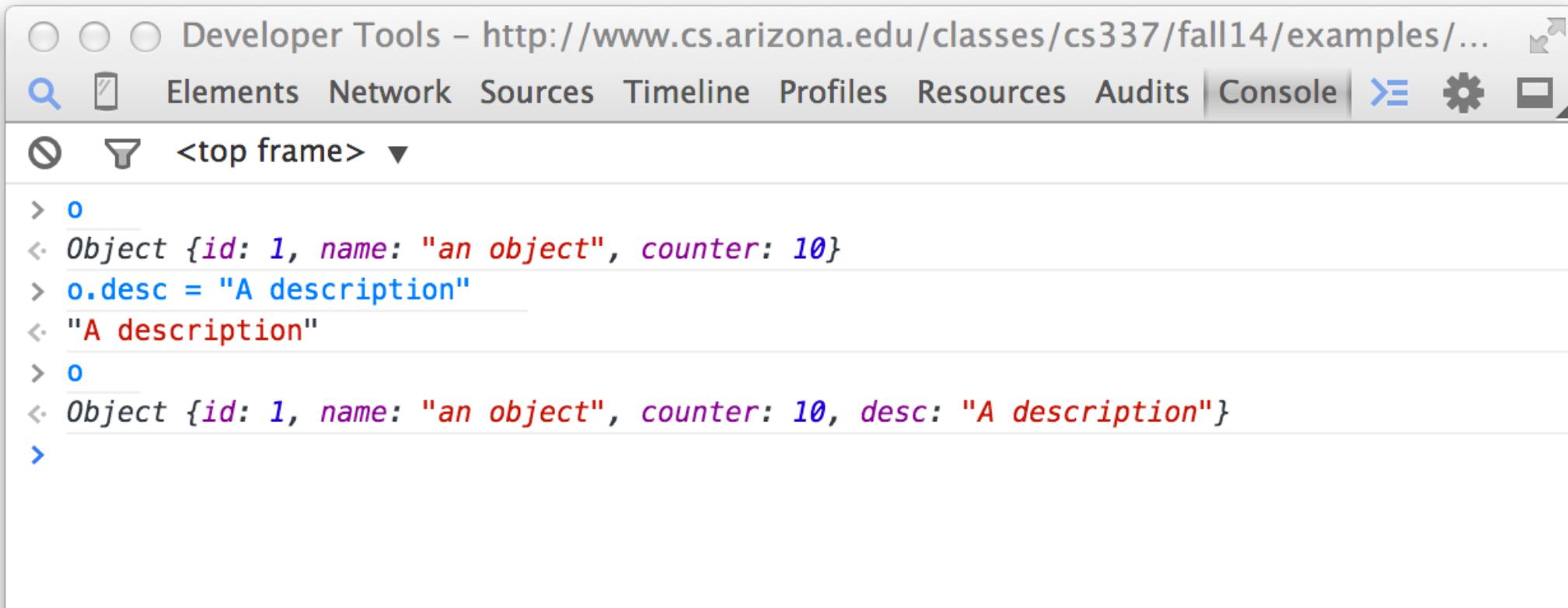
The screenshot shows a browser's developer tools console interface. The title bar reads "Developer Tools – http://www.cs.arizona.edu/classes/cs337/fall14/exampl". The menu bar includes "Elements", "Network", "Sources", "Timeline", "Profiles", "Resources", "Audits", "Console", and "Console" (with a dropdown arrow). The main area shows the following interaction:

```
> o
< Object {id: 1, name: "an object", counter: 10}
> o["id"]
< 1
> typeof(o["name"])
< "string"
>
```

The "name" property is highlighted in red, indicating it is a string value.

Objects

- Assigning to undefined properties creates them



The screenshot shows the Google Chrome Developer Tools interface with the 'Console' tab selected. The title bar indicates the developer tools are running on a local host example page. The console output shows the following sequence of commands and their results:

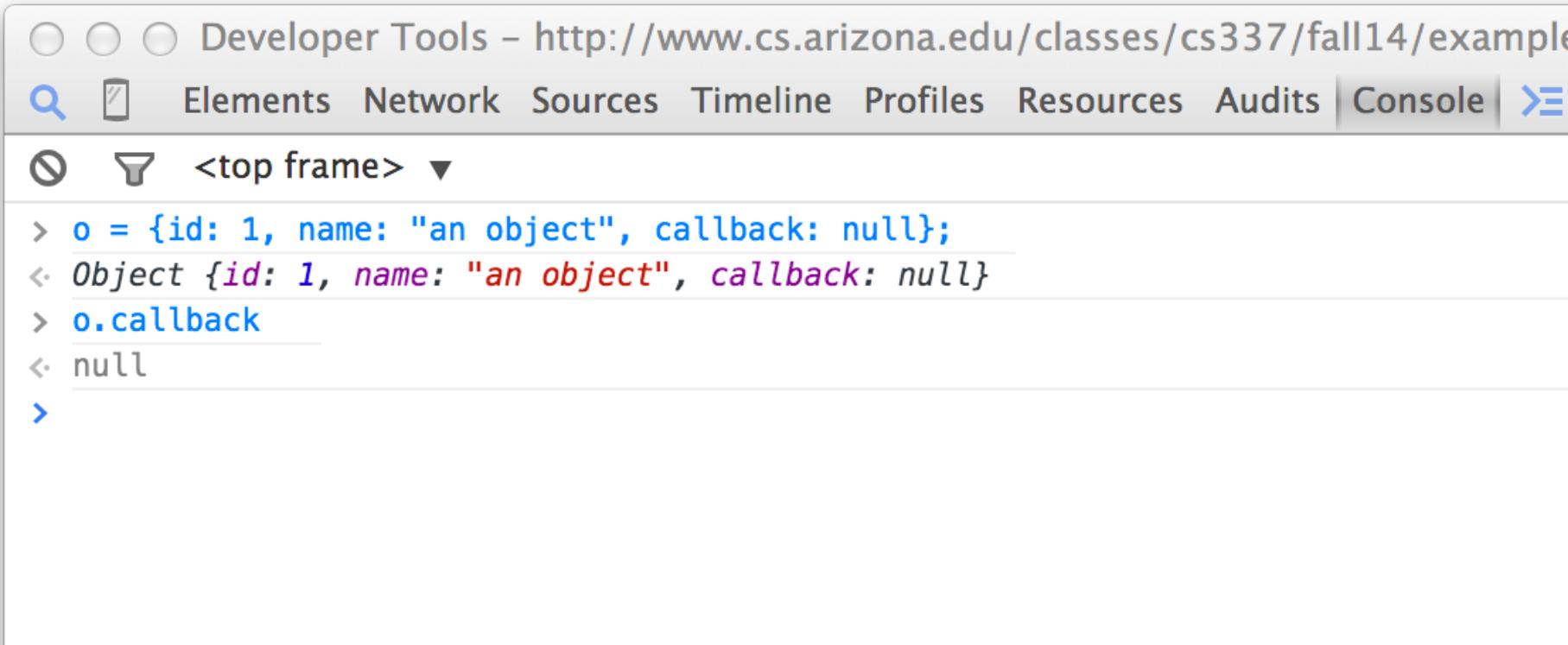
```
> o
< Object {id: 1, name: "an object", counter: 10}
> o.desc = "A description"
< "A description"
> o
< Object {id: 1, name: "an object", counter: 10, desc: "A description"}
```

The first command creates an object with three properties: id (1), name ("an object"), and counter (10). The second command adds a new property desc with the value "A description". The third command retrieves the object again, showing it now has four properties: id, name, counter, and desc.

null

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's developer tools console window. The title bar reads "Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/example". The tabs at the top are Elements, Network, Sources, Timeline, Profiles, Resources, Audits, and Console. The "Console" tab is active. Below the tabs, there are two icons: a magnifying glass and a funnel. The text area shows the following interaction:

```
> o = {id: 1, name: "an object", callback: null};  
< Object {id: 1, name: "an object", callback: null}  
> o.callback  
< null  
>
```

Operators

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

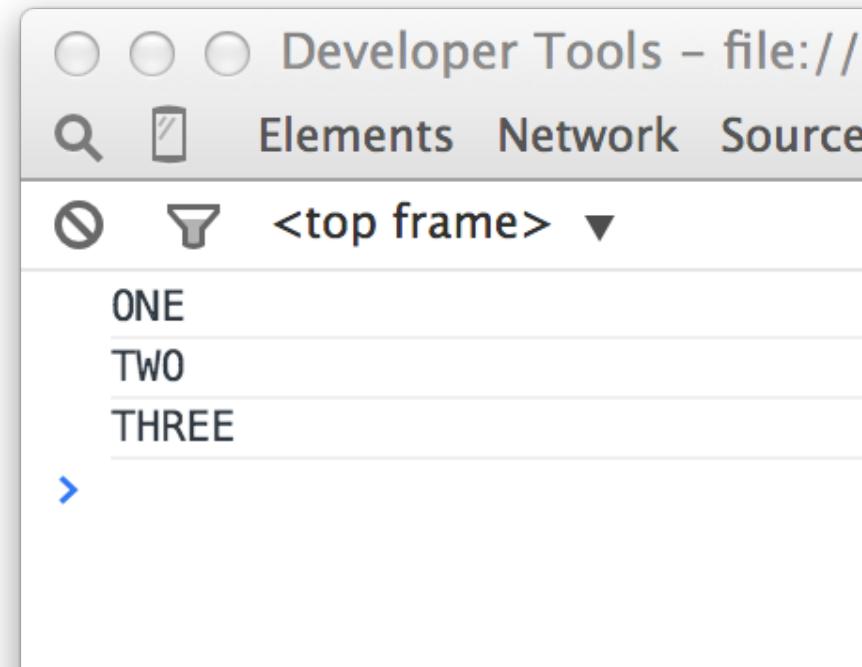
Control Structures

- `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

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

- 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.



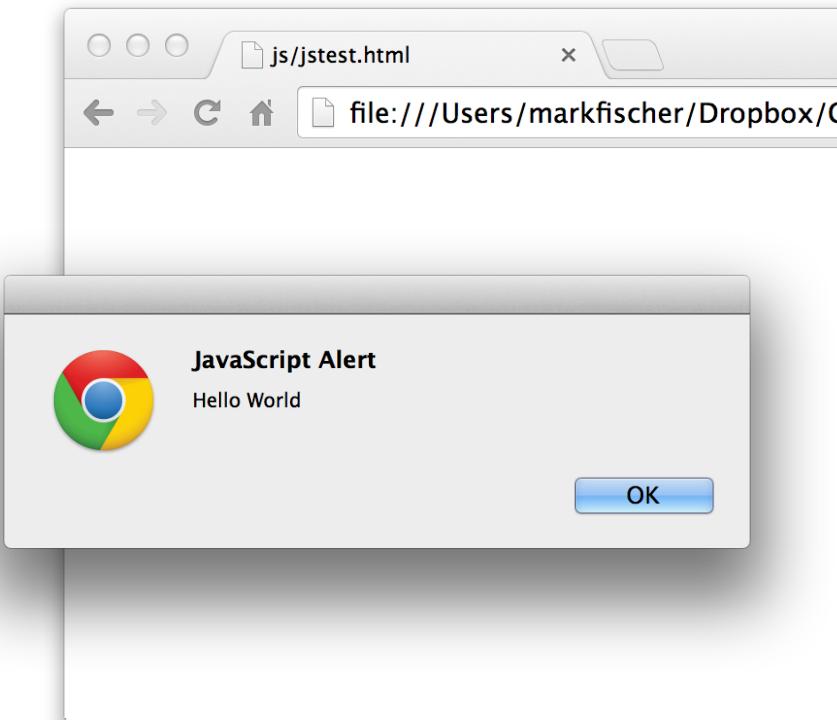
Basic I/O

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

alert()

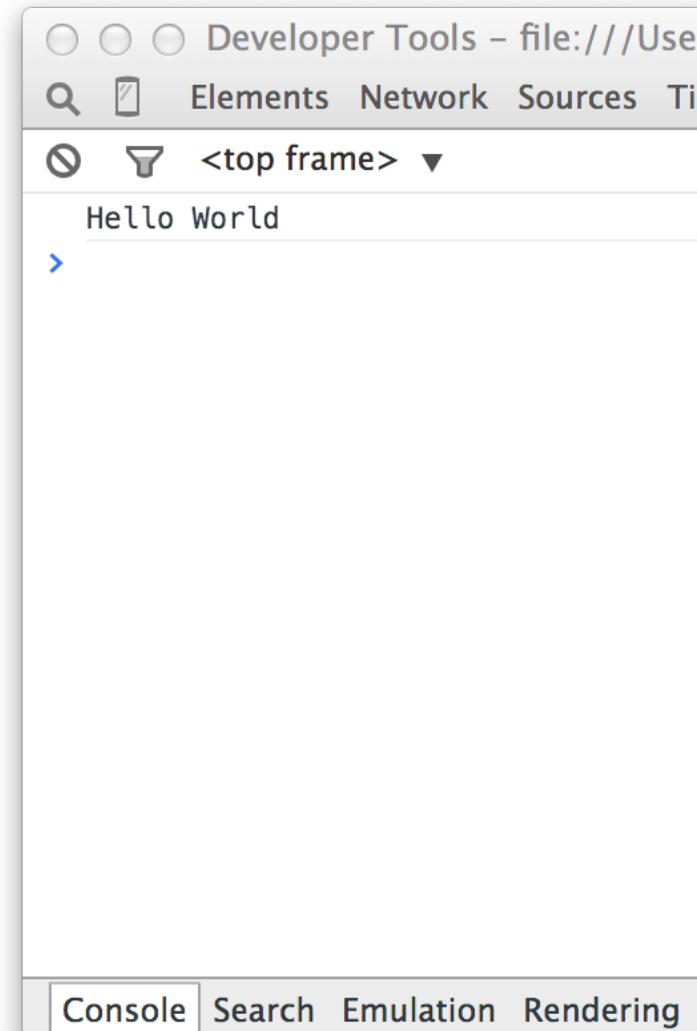
```
alert("Hello World");
```

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



console.log()

```
console.log("Hello World");
```

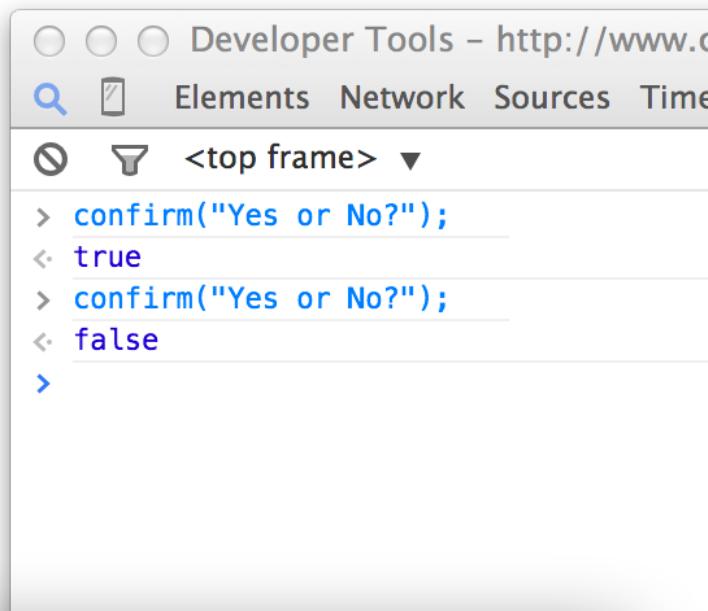


- Quick way to get some debugging out.
- Doesn't block execution, so usually a better choice for debugging and testing than `alert()`.

confirm()

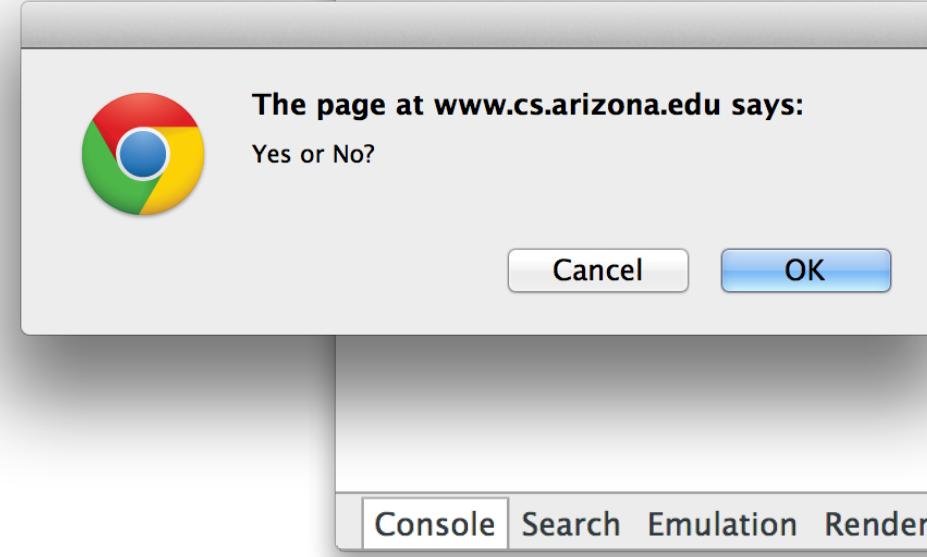
```
confirm("Yes or No?");
```

- Ask for a **true** or **false** response from the user.



The screenshot shows a browser developer tools console window titled "Developer Tools – http://www.cs.arizona.edu". The console tab is selected. The code entered is:

```
> confirm("Yes or No?");
<- true
> confirm("Yes or No?");
<- false
>
```



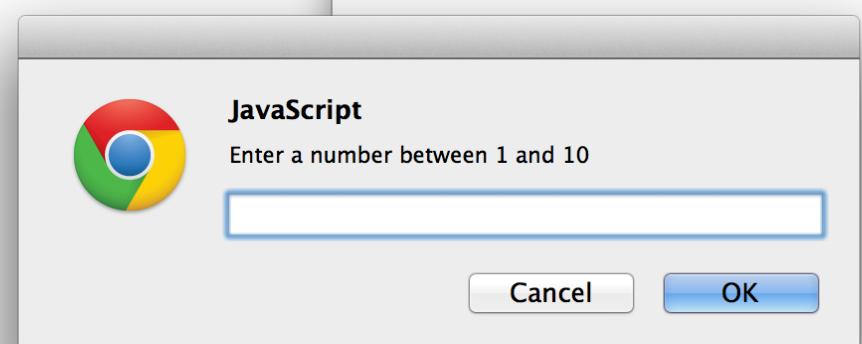
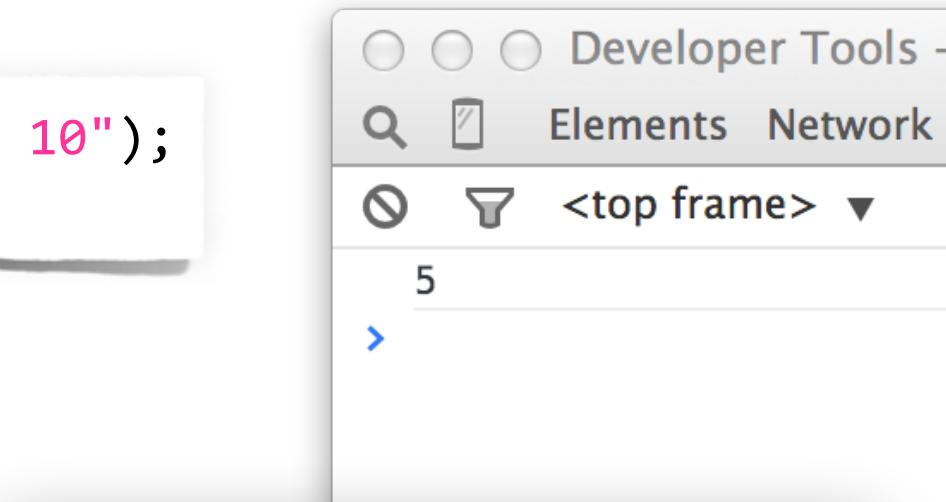
A Chrome browser window is displayed, showing a confirmation dialog box. The dialog box contains the text "The page at www.cs.arizona.edu says:" followed by "Yes or No?". At the bottom are two buttons: "Cancel" and "OK".

At the bottom of the slide, there is a navigation bar with tabs: Console, Search, Emulation, and Rendering. The "Console" tab is currently active.

prompt()

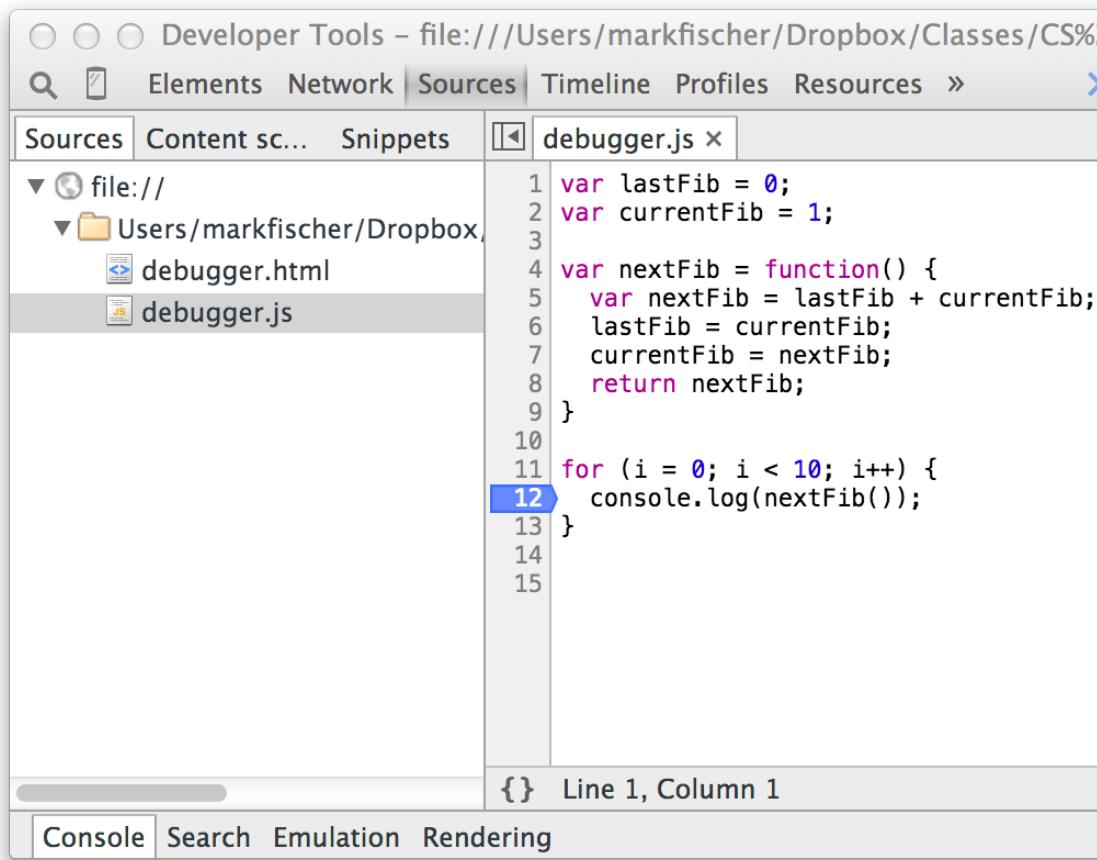
```
prompt("Enter a number between 1 and 10");  
console.log(i);
```

- Ask for user input as a text string.



Debugger

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



The screenshot shows the 'Sources' tab in the Chrome Developer Tools. The left sidebar lists files: 'file:///' and 'Users/markfischer/Dropbox/'. Under 'Dropbox', there are two files: 'debugger.html' and 'debugger.js'. 'debugger.js' is currently selected. The main pane displays the code for 'debugger.js':

```
1 var lastFib = 0;
2 var currentFib = 1;
3
4 var nextFib = function() {
5     var nextFib = lastFib + currentFib;
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
```

A blue arrow points to line 12, indicating it is the current line of execution. At the bottom right of the code area, it says 'Line 1, Column 1'. Below the code area are tabs for 'Console', 'Search', 'Emulation', and 'Rendering'.

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



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



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



```
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

- 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;
}
```

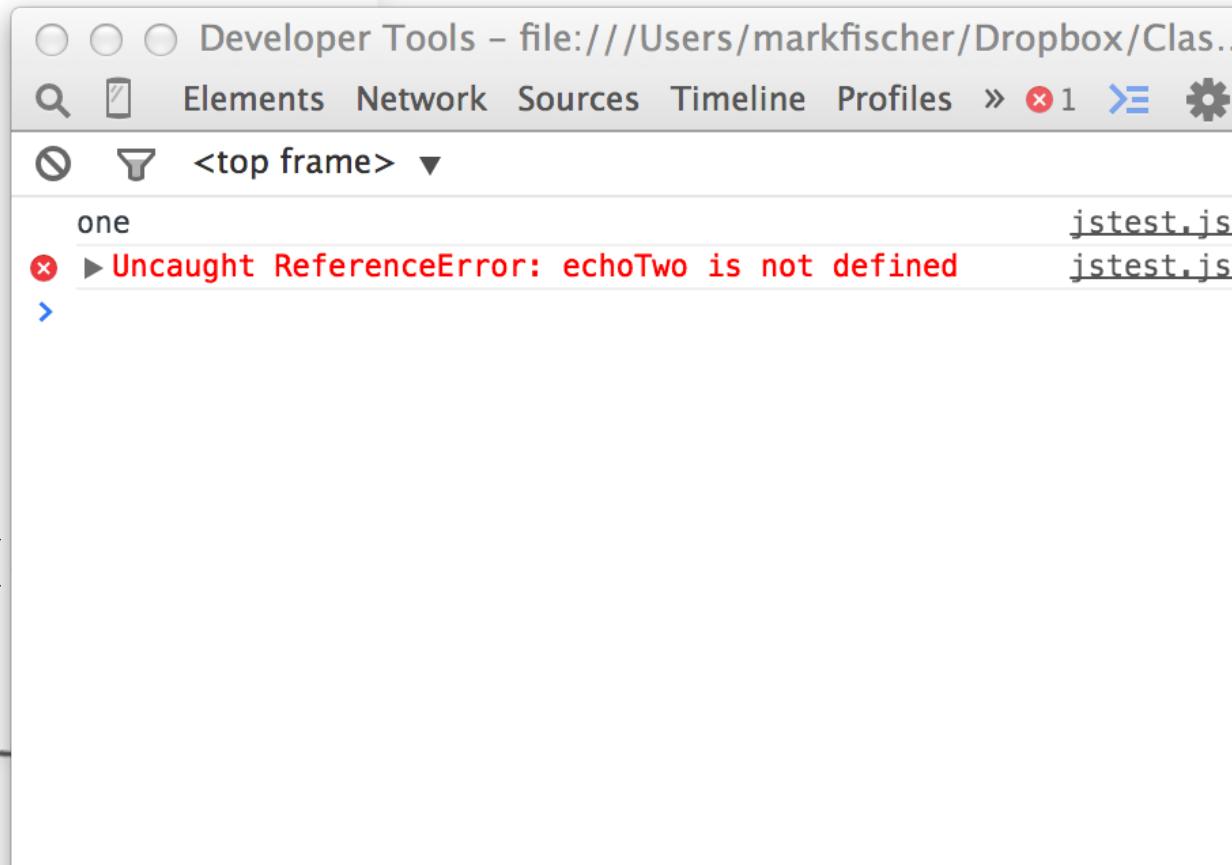
Functions

```
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(
  return a;
}
```



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.

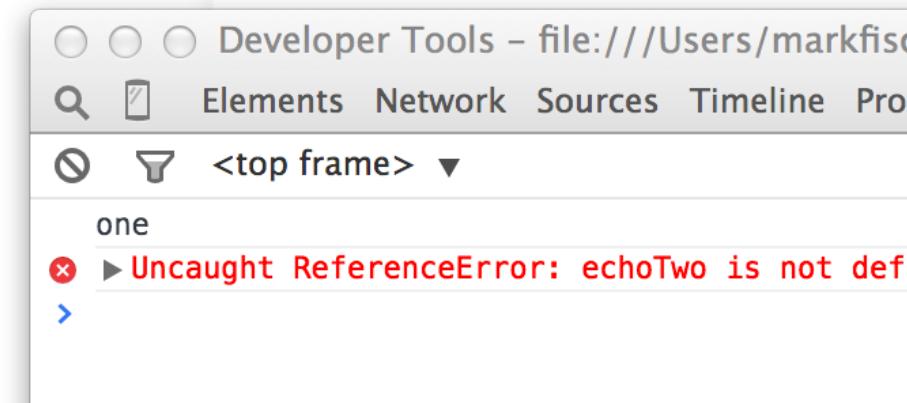
javascriptweblog.wordpress.com/2010/07/06/function-declarations-vs-function-expressions/

```
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

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

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

Functions

- What is the console output here?

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

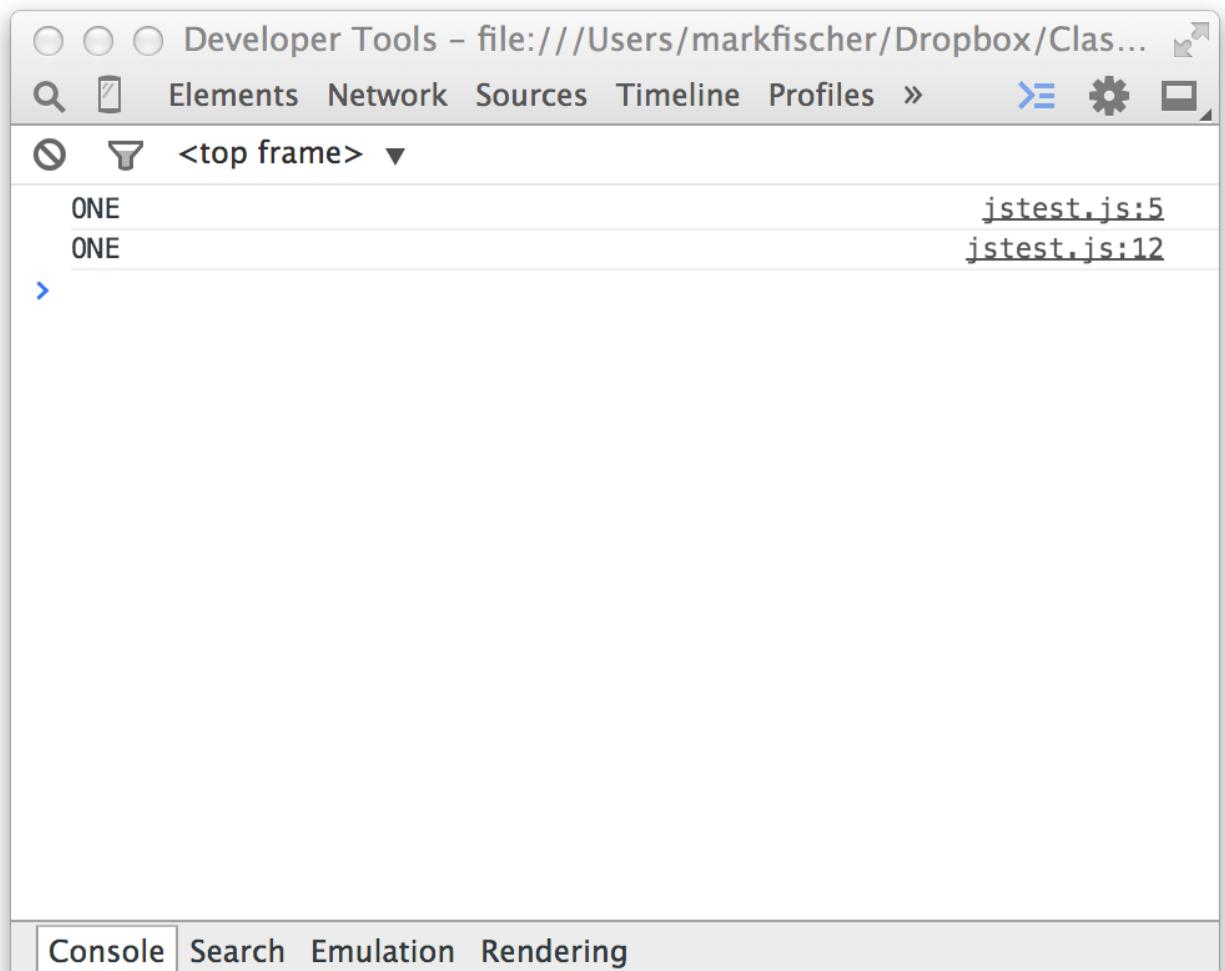
```
console.log( echo("one") );
```

```
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.



The screenshot shows the Google Chrome Developer Tools interface, specifically the Sources tab. The title bar reads "Developer Tools - file:///Users/markfischer/Dropbox/Clas...". The main content area displays two function declarations:

```
ONE
ONE
```

The first declaration is associated with the line number "jstest.js:5" and the second with "jstest.js:12". At the bottom of the panel, there are tabs for "Console", "Search", "Emulation", and "Rendering", with "Console" being the active tab.

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

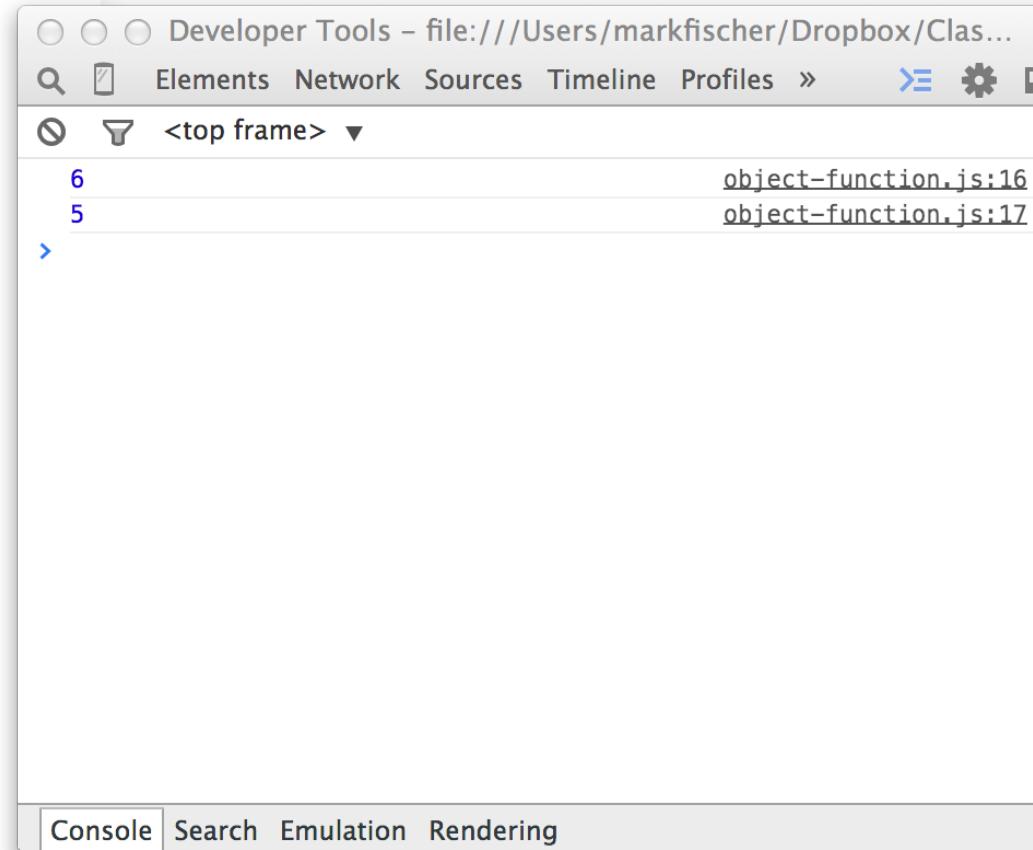
- Can be used anywhere an expression is valid.
 - Can be more flexible because of this.
 - Are evaluated and assigned at run time.

Objects and Functions

- Functions can be added to objects as property variables.
- Many 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.

```
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) );
```

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() );
```

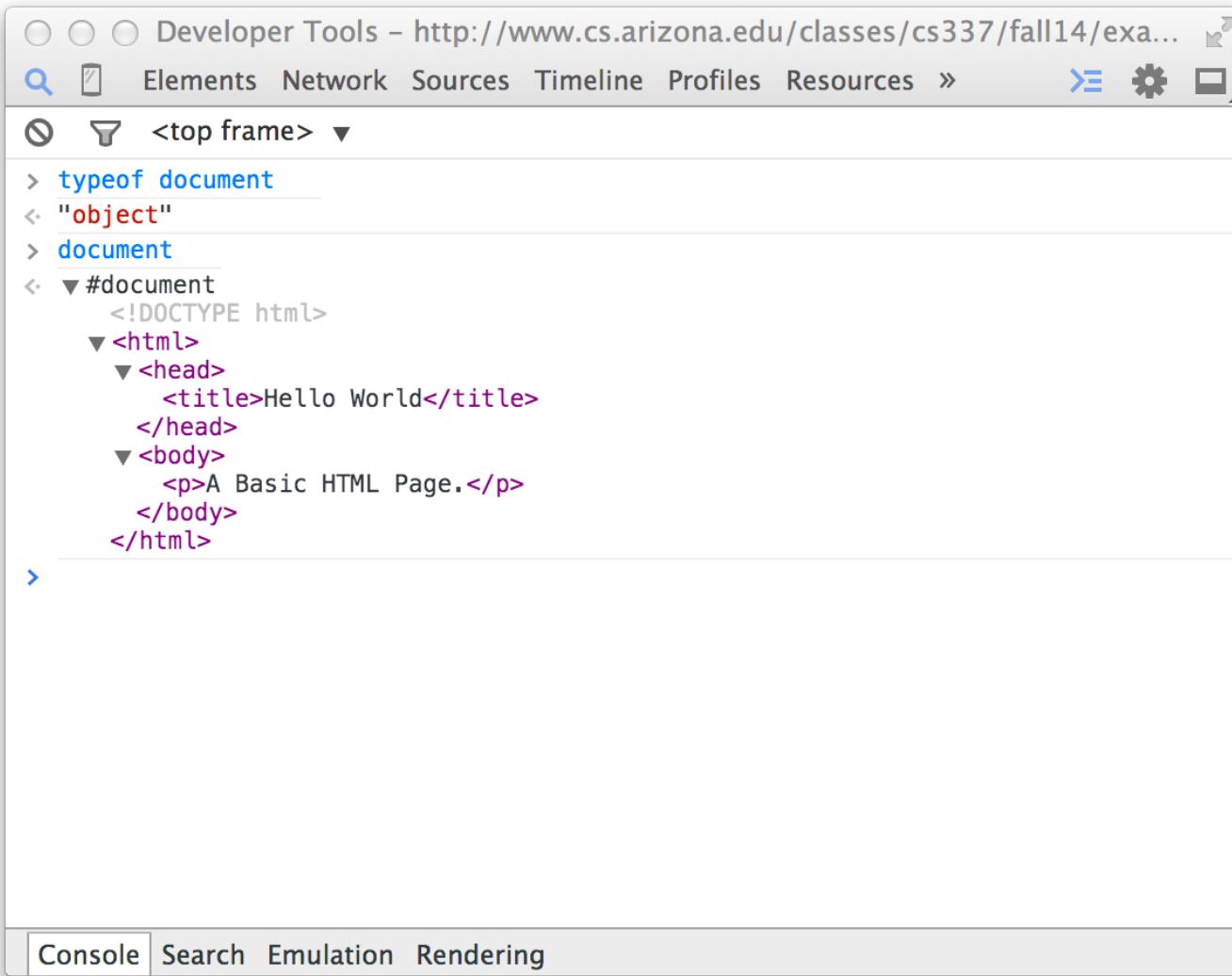
The document Object

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

The document Object

- 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

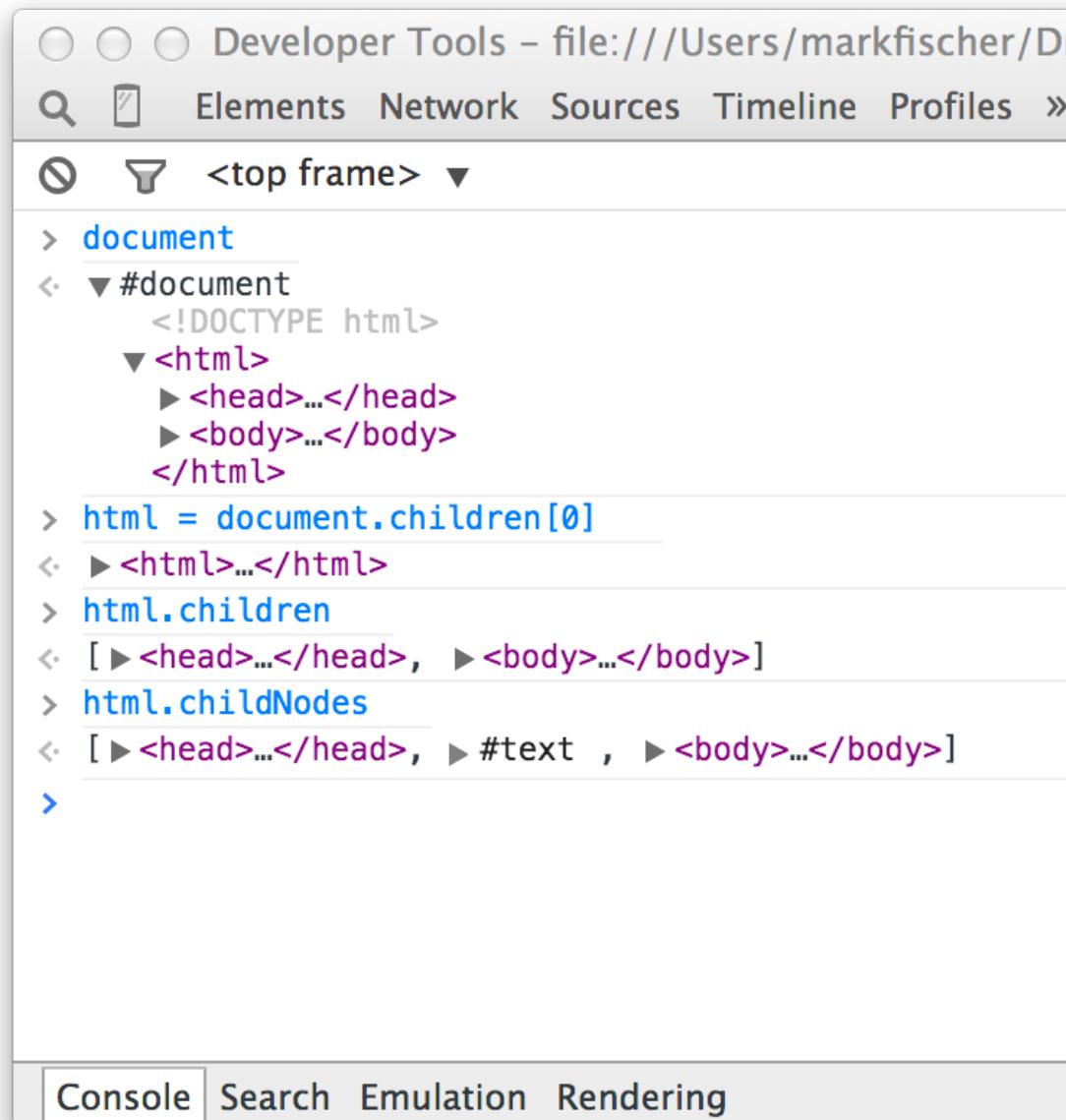


The screenshot shows the "Elements" tab of a browser's developer tools. The title bar indicates "Developer Tools - http://www.cs.arizona.edu/classes/cs337/fall14/exa...". The main pane displays the DOM tree of a basic HTML page titled "Hello World". The tree starts with the global variable `document`, which is an `object`. The `document` object contains the `#document` node, which represents the entire HTML document. This node contains the `html`, `head`, and `body` elements. The `head` element contains a `title` element with the text "Hello World". The `body` element contains a single `p` element with the text "A Basic HTML Page.". At the bottom of the developer tools interface, there is a navigation bar with tabs for "Console", "Search", "Emulation", and "Rendering".

```
> typeof document
<- "object"
> document
<- ▼#document
    <!DOCTYPE html>
    ▼<html>
        ▼<head>
            <title>Hello World</title>
        </head>
        ▼<body>
            <p>A Basic HTML Page.</p>
        </body>
    </html>
>
```

The document Object

- The document object represents the root element of our DOM tree.
- It has child nodes, and each node has various attributes.
- Note the difference between `.children` and `.childNodes`



The screenshot shows the Chrome Developer Tools interface. The title bar reads "Developer Tools - file:///Users/markfischer/D". The top navigation bar includes "Elements" which is currently selected, along with "Network", "Sources", "Timeline", and "Profiles". Below the navigation is a search bar and a dropdown menu set to "<top frame>". The main area displays a hierarchical DOM tree. At the top is the "document" node, followed by the "#document" node which contains the root "

" declaration and a single "" element. This "" element has two children: a "" element and a "" element. In the console below the tree, several lines of code are shown:

```
> document
<- ▼#document
    <!DOCTYPE html>
    ▼<html>
        ▶<head>...</head>
        ▶<body>...</body>
    </html>
> html = document.children[0]
<- ▶<html>...</html>
> html.children
<- [▶<head>...</head>, ▶<body>...</body>]
> html.childNodes
<- [▶<head>...</head>, ▶#text , ▶<body>...</body>]
>
```

The console also shows the current selection in the tree is the "" node, indicated by blue highlighting around the code "html = document.children[0]" and the node itself in the tree view.

The document Object

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

```
Developer Tools - file:///Users/markfischer/Dr
Elements Network Sources Timeline Profiles >
<top frame> ▾
> document
<- #document
    <!DOCTYPE html>
    ▶<html>...</html>
> html = document.children[0]
<- ▶<html>...</html>
> html.innerHTML
<- "<head>
    <title>Hello World</title>
</head>
<body>
    <p>A Basic HTML Page.</p>
</body>""
> typeof html
<- "object"
>
```

Console Search Emulation Rendering

The document Object

- The **document** object is *NOT* part of the Javascript language.
- 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

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")`

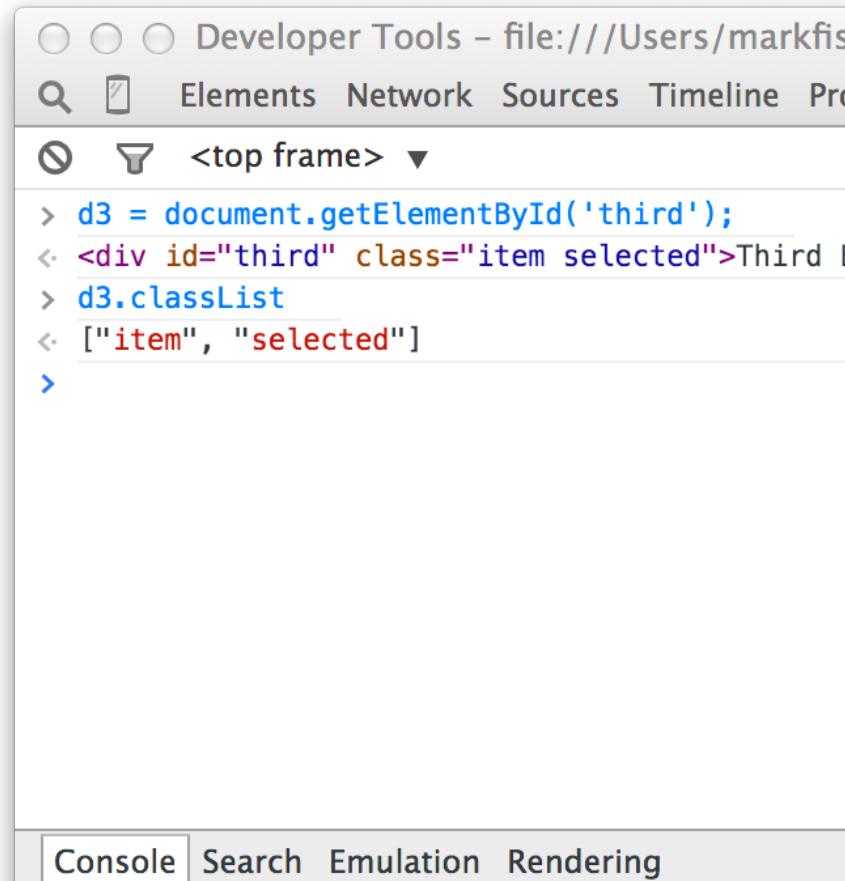
getElementById

- Gets an HTMLElement 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>
  <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 selected">
      Third Block
    </div>
  </div>
</body>
</html>
```



The screenshot shows the Chrome Developer Tools interface with the "Console" tab selected. The console output displays the following code and its execution results:

```
d3 = document.getElementById('third');
<div id="third" class="item selected">Third Block
d3.classList
["item", "selected"]
```

The "selected" class is highlighted in red in the original HTML code, and it is also highlighted in red in the "classList" output, indicating it was successfully selected.

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>

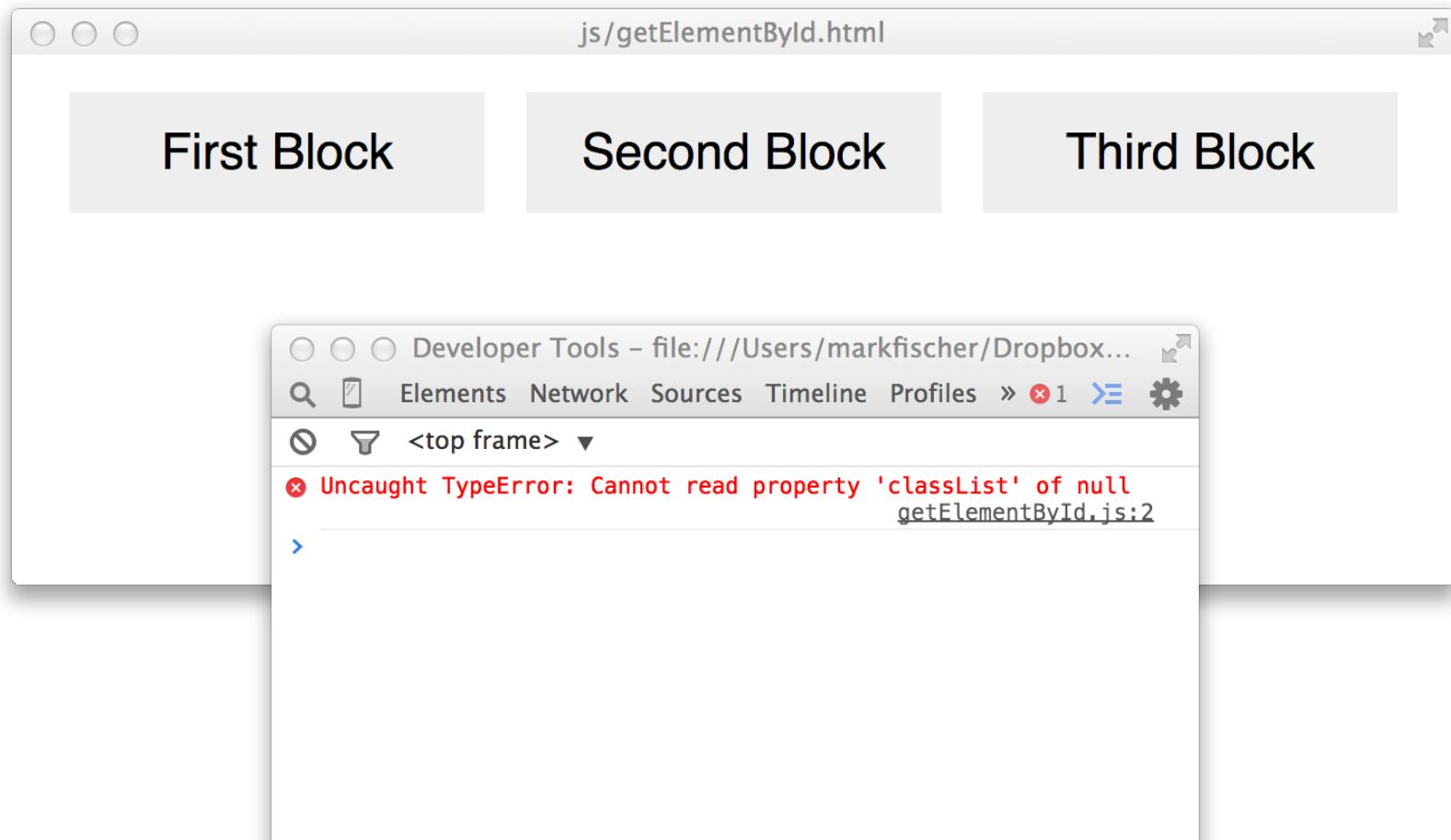
<body>
  <div id="main">
    <div id="first" class="item">First Block</div>
    <div id="second" class="item">Second Block</div>
    <div id="third" class="item">Third Block</div>
  </div>
</body>
</html>
```

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



Updating the DOM

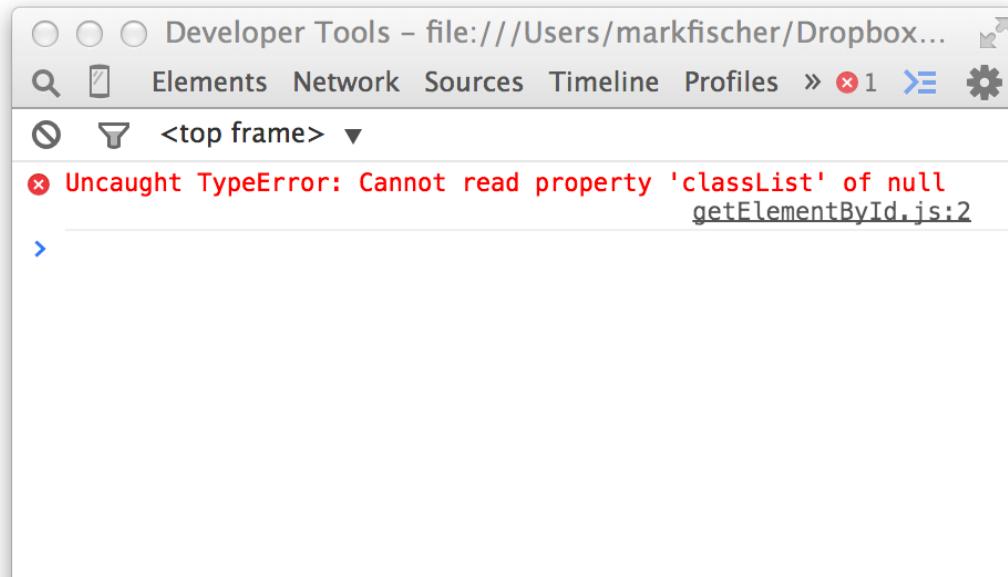
- Hmm nothing happened. Why? Check the console.



Updating the DOM

- Uncaught TypeError: Cannot read property 'classList' 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?

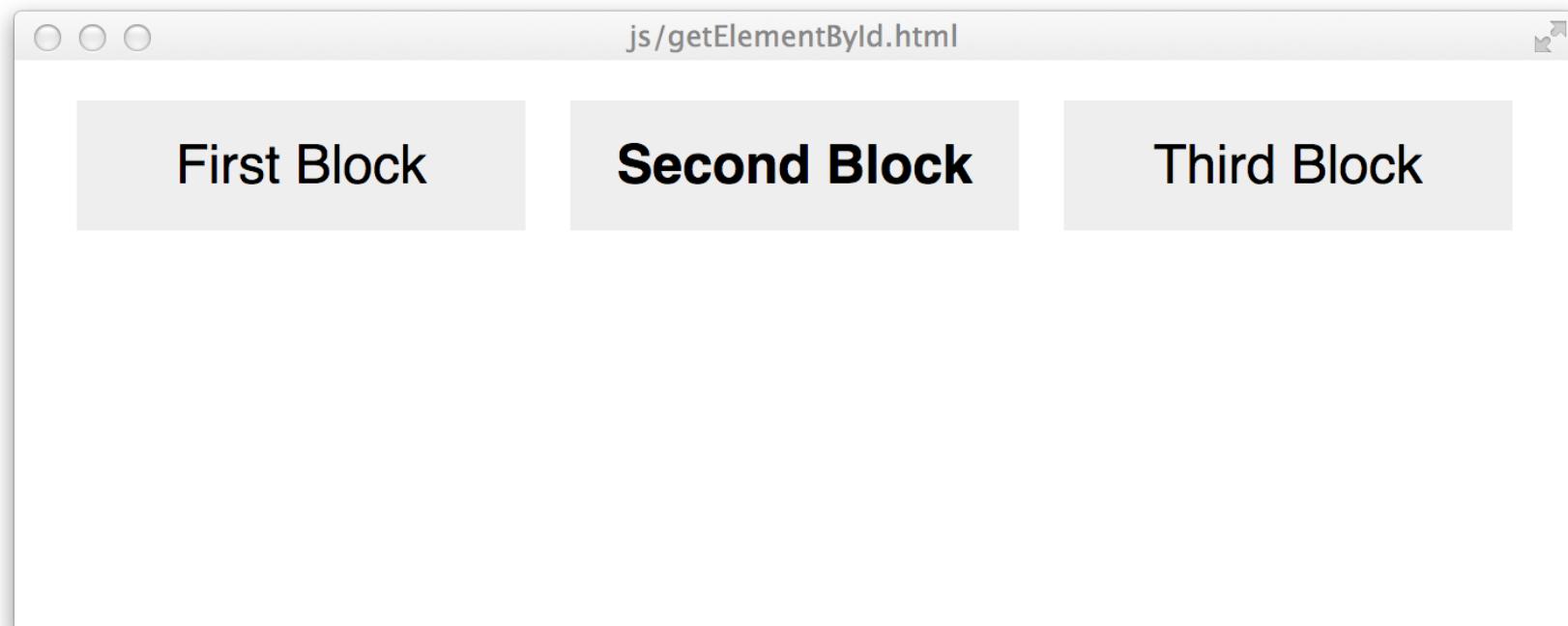
```
<!doctype html>
<head>
  <title>js/getElementById.html</title>
  <link rel="stylesheet" type="text/css"
        href="getElements.css" />
</head>

<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, it's 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 (function) 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.

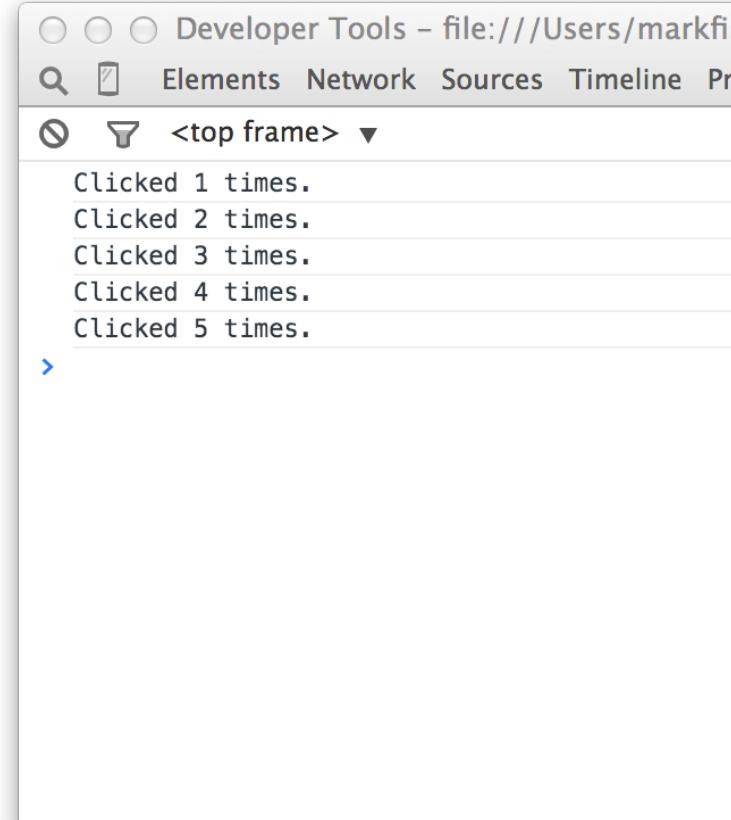
Events

- 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>
```



Events

- Is it really that simple? What about IE, doesn't that always mess us up?
- Well, yes. Of course it does.
- *object.addEventListener()* didn't come to IE until 9
- Earlier methods for adding event listeners were directly in markup, or via *object.event = callback;*

```
<a href="#" onclick="callbackName">Link</a>
```

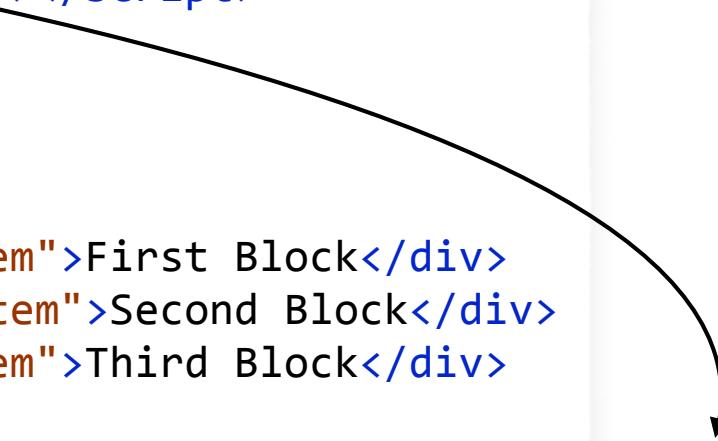
window load Event

- 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>

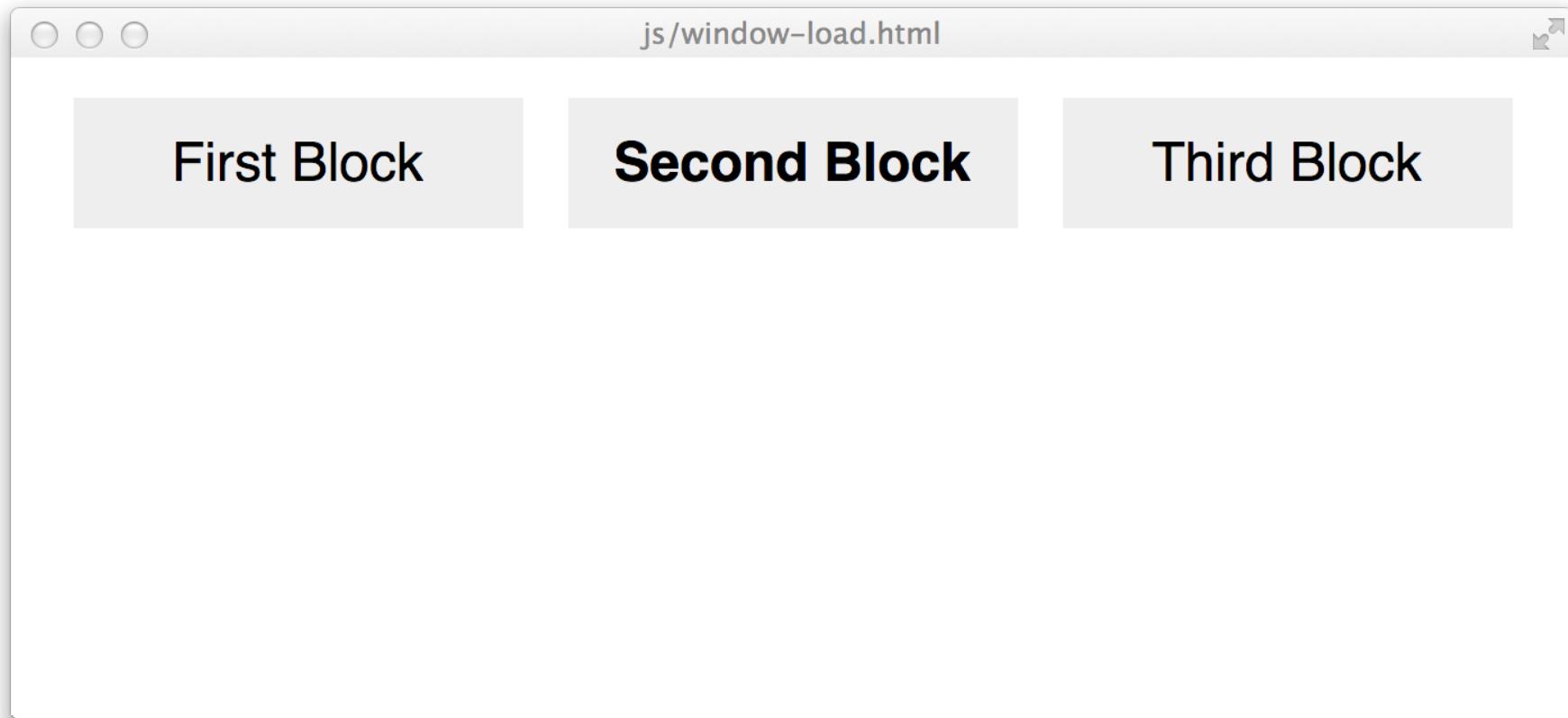
<body>
  <div id="main">
    <div id="first" class="item">First Block</div>
    <div id="second" class="item">Second Block</div>
    <div id="third" class="item">Third Block</div>
  </div>
</body>
</html>
```



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

window load Event

- Works!



window load Event

- Since addEventListener doesn't work with IE 8 or older, to provide a more robust solution you'd have to do browser capabilities detection.

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

window load Event

- IE 8 supported a different method, the `object.attachEvent` method.
- Even older browsers only support a single “onload” property.
- If only someone would write a library that did all this for us...

```
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 curronload = window.onload;
      var newonload = function() {
        curronload();
        myFunciton();
      };
      window.onload = newonload;
    } else {
      window.onload = myFunciton;
    }
  }
}
```

Putting Pieces Together



Demo

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>

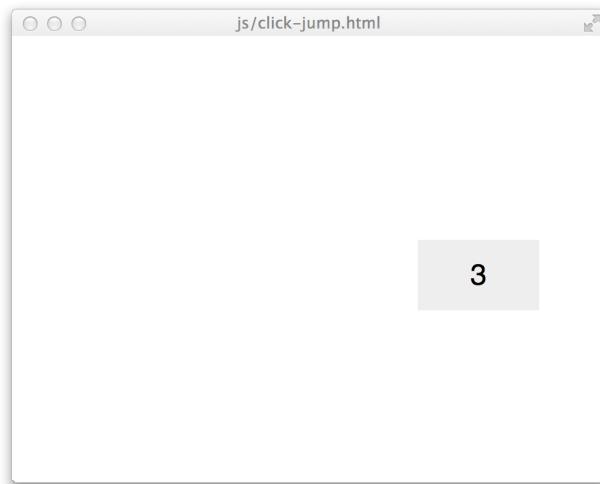
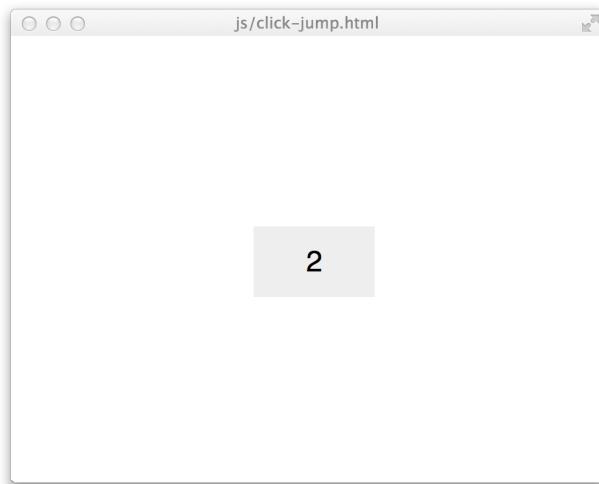
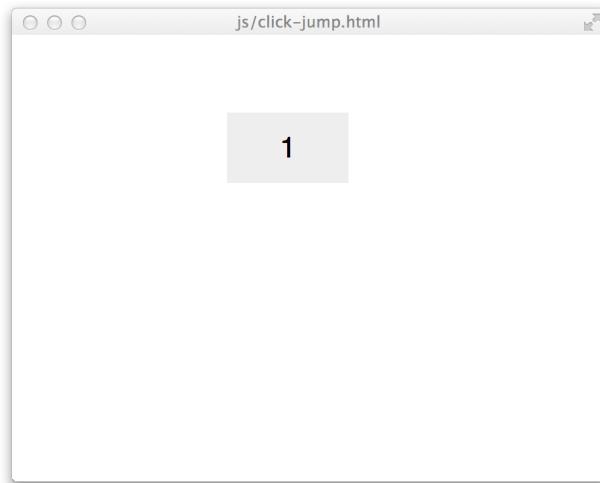
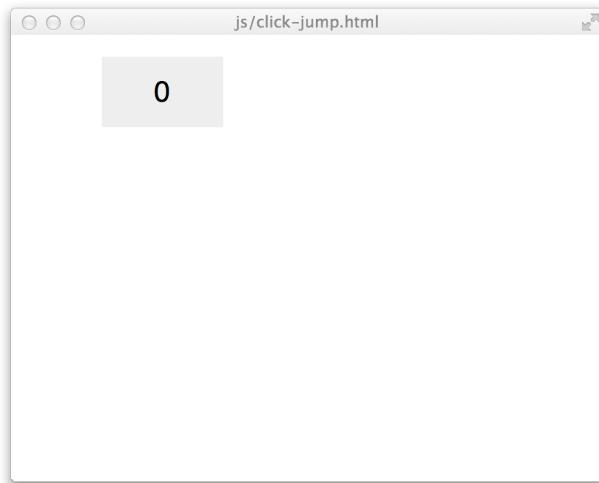
<body>
  <div id="main">
  </div>
</body>
</html>
```

click-count.html

```
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);
  }
});
```

click-jump.html



click-jump.html

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

    if (curCount == 1) {
        this.style.position = "absolute";
    }

    var max_x = window.innerWidth - 110;
    var max_y = window.innerHeight - 60;
    var newX = Math.random() * max_x;
    var newY = Math.random() * max_y;
    newX = Math.floor(newX);
    newY = Math.floor(newY);

    this.style.top = newY.toString() + "px";
    this.style.left = newX.toString() + "px";
}
```

Inspiration

- Case study on copying stuff from other people.
- <https://account.arizona.edu/welcome>

Updating Styles

- Many ways of updating DOM elements involves changing its CSS Style Attributes
 - Positioning an element somewhere
 - Changing font styles
 - Changing colors, borders, etc

element.style

```
element.style.color = "#cccccc";
```

- An HTMLElement object has a style property
- The style property is itself an object, having properties for all the CSS properties appropriate to that element

element.style

```
size = 2;  
element.style.fontSize = size.toString() + "em";
```

- All values are strings. If you have to assign numerical values, you need to convert them to strings.
- You also need to make sure the value is a complete and valid value for the property, including any units

element.style

```
element.style.fontSize = "2em";
```

- You can't have javascript variable names that contain a dash... its the subtraction operator.
- CSS properties with dashes in their name become camelCased

```
font-size    ->    fontSize
```

Timing Events

- Browsers implement Javascript in a threaded environment.
- Events can be queued to fire at a later time.
- `window.setTimeout()`
- `window.setInterval()`

<https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers>

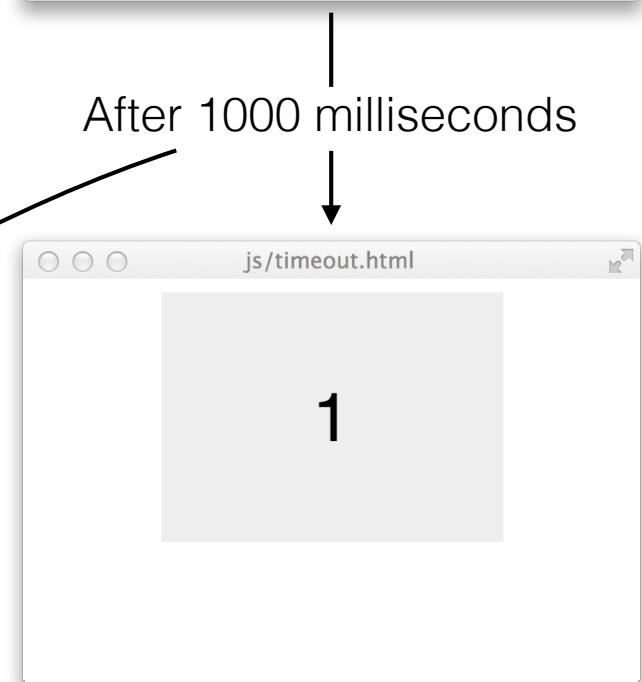
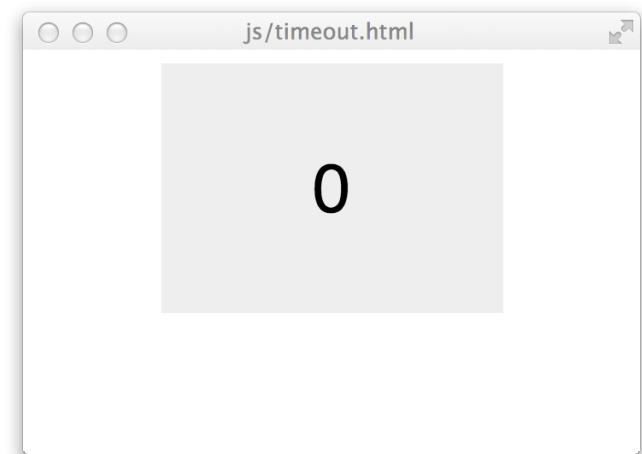
setTimeout()

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

<body>
  <div id="main">0</div>

  <script>
    var counter = function() {
      var d = document.getElementById('main');
      var curCount = Number(d.textContent);
      curCount++;
      d.textContent = curCount.toString();
    }

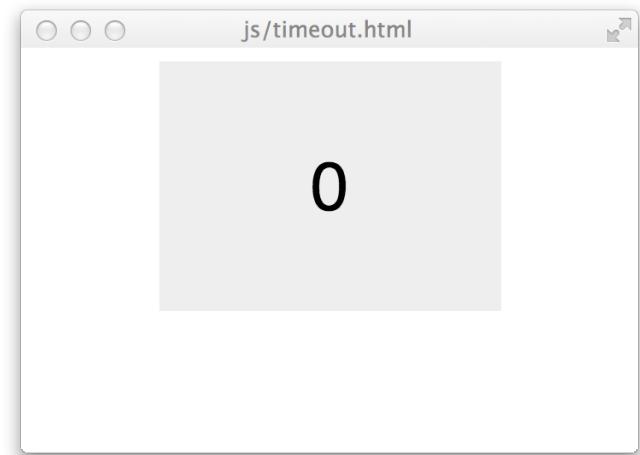
    window.setTimeout(counter, 1000);
  </script>
</body>
</html>
```



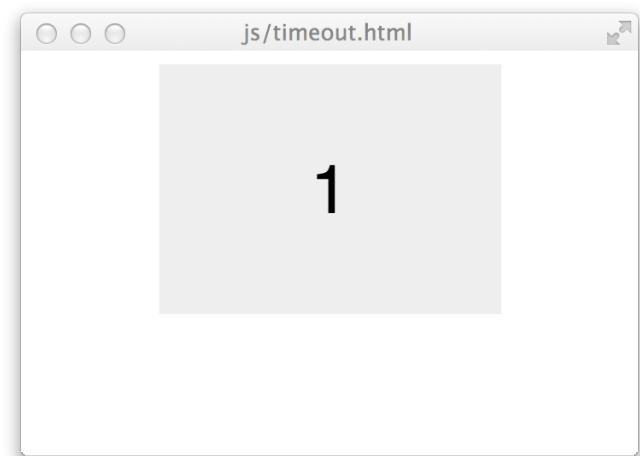
After 1000 milliseconds

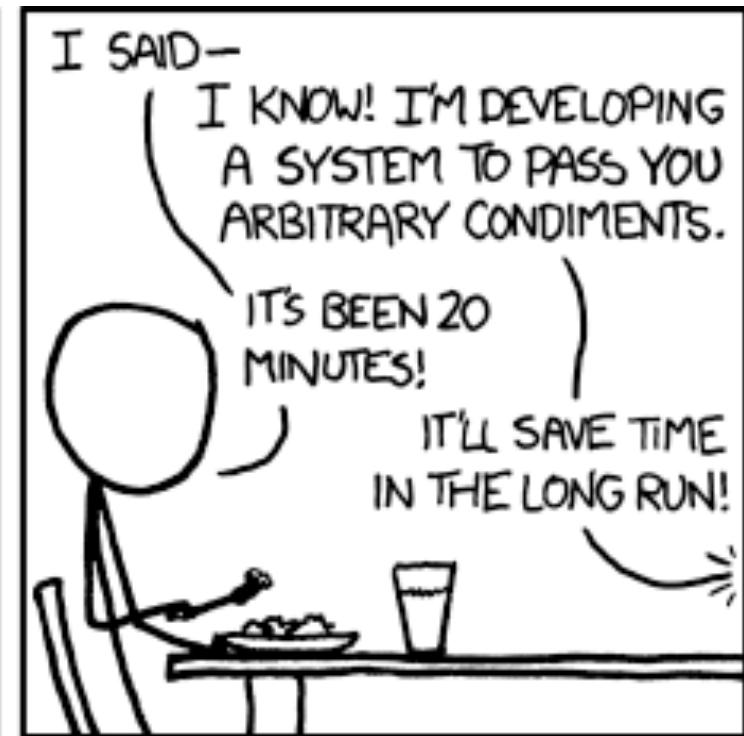
setInterval()

- `setTimeout()` only fires a single time.
- To fire on an interval, use `setInterval()`, or continually call `setTimeout()`.
- Demo



After 1000 milliseconds
↓

A vertical line with a downward arrow pointing from the text "After 1000 milliseconds" to the second screenshot.



Classes

Oops, sorry, there are no classes.

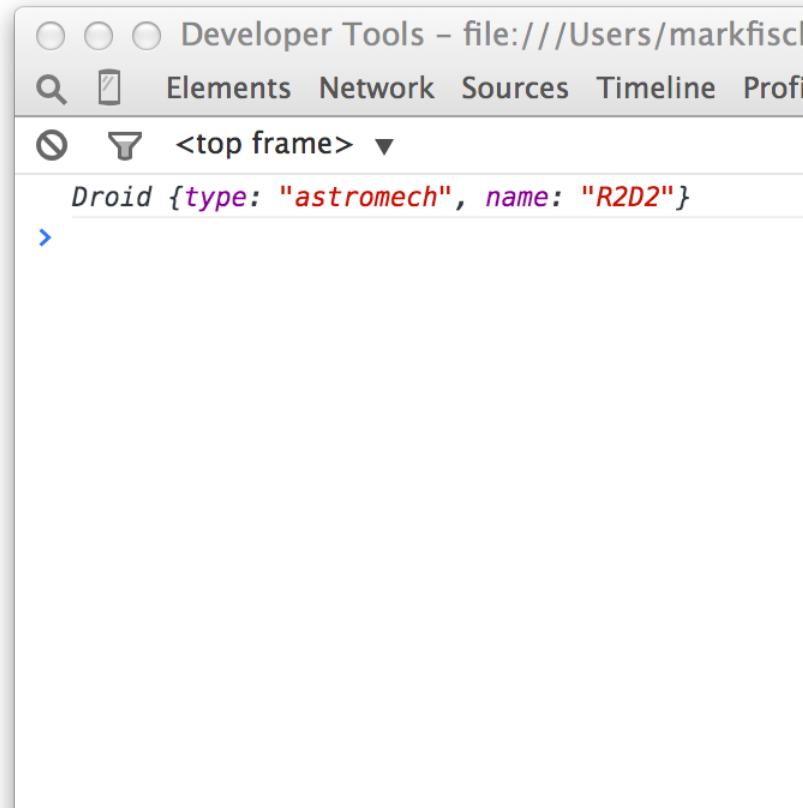
Class Like Thingies

- 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.

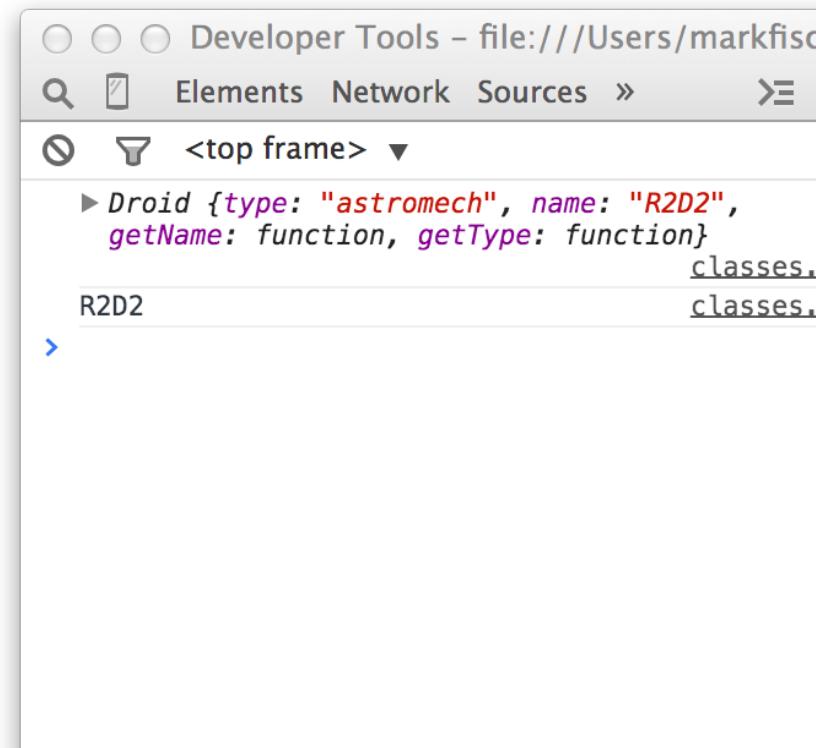
```
function Droid(type, name) {  
  this.type = type;  
  this.name = name;  
}  
  
var r2 = new Droid('astromech', 'R2D2');  
var c3 = new Droid('protocol', 'C3PO');  
  
console.log(r2);
```



prototypes

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

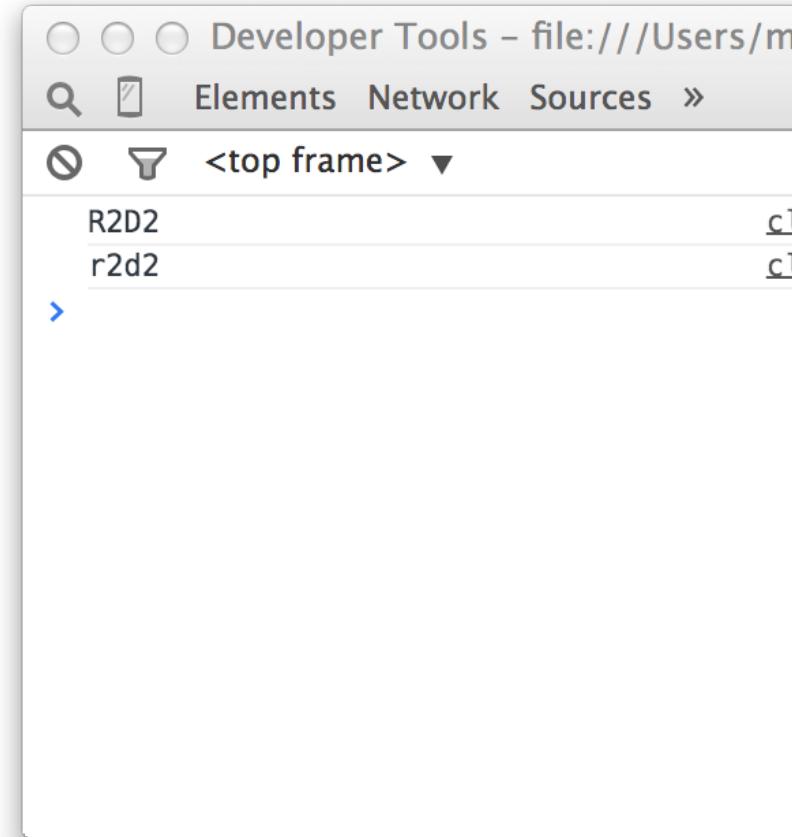
```
function Droid(type, name) {  
  this.type = type;  
  this.name = name;  
}  
  
Droid.prototype = {  
  getName: function() { return this.name },  
  getType: function() { return this.type }  
}  
  
var r2 = new Droid('astromech', 'R2D2');  
var c3 = new Droid('protocol', 'C3PO');  
  
console.log(r2);  
console.log(r2.getName());
```



prototypes

- 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());
```



myQuery

- jQuery is a very popular Javascript toolkit which abstracts away some of the underlying complexity.
- Can we build our own simple toolkit?
- Of course we can...
- jQuery doesn't own \$

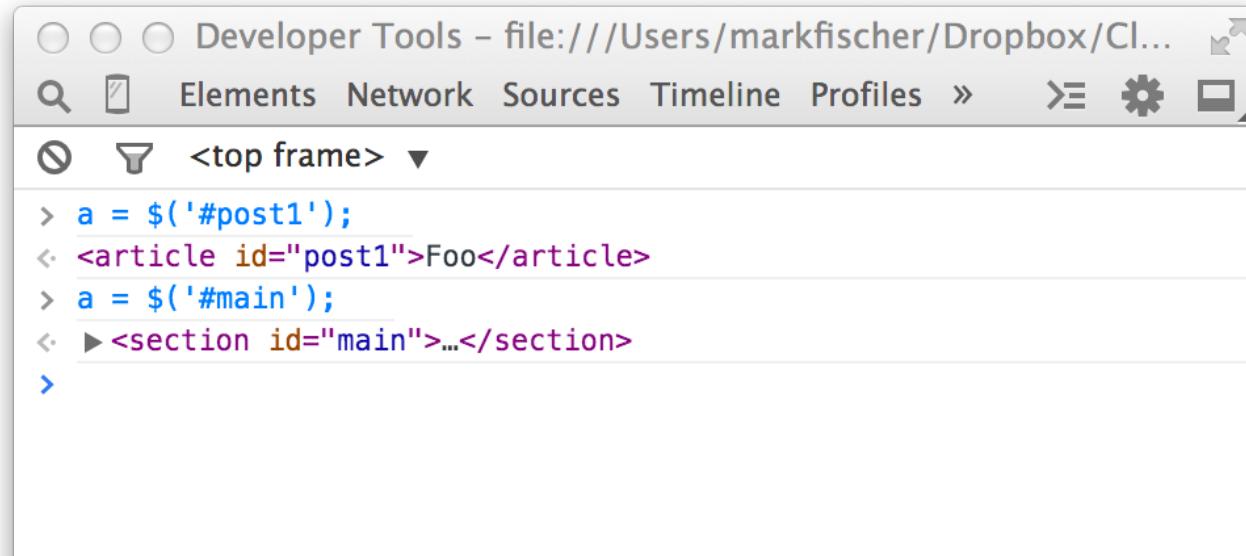
Basic Selection

- Using `document.getElementById()` isn't too bad, but it sure is a lot of typing.
- Can we use the `$('selector')` pattern?

```
var $ = function myQuery(selector) {  
    // See if selector starts with a #. If so we're looking for an ID  
    if (selector[0] == '#') {  
        // Strip off the # sign  
        var selector = selector.substring(1, selector.length);  
        var element = document.getElementById(selector);  
        return element;  
    }  
}
```

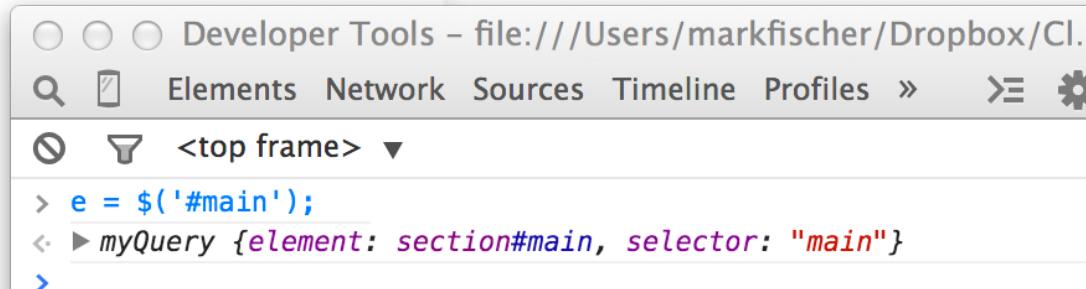
Basic Selection

```
var $ = function myQuery(selector) {
  // See if selector starts with a #. If so we're looking for an ID
  if (selector[0] == '#') {
    // Strip off the # sign
    var selector = selector.substring(1, selector.length);
    var element = document.getElementById(selector);
    return element;
  }
}
```



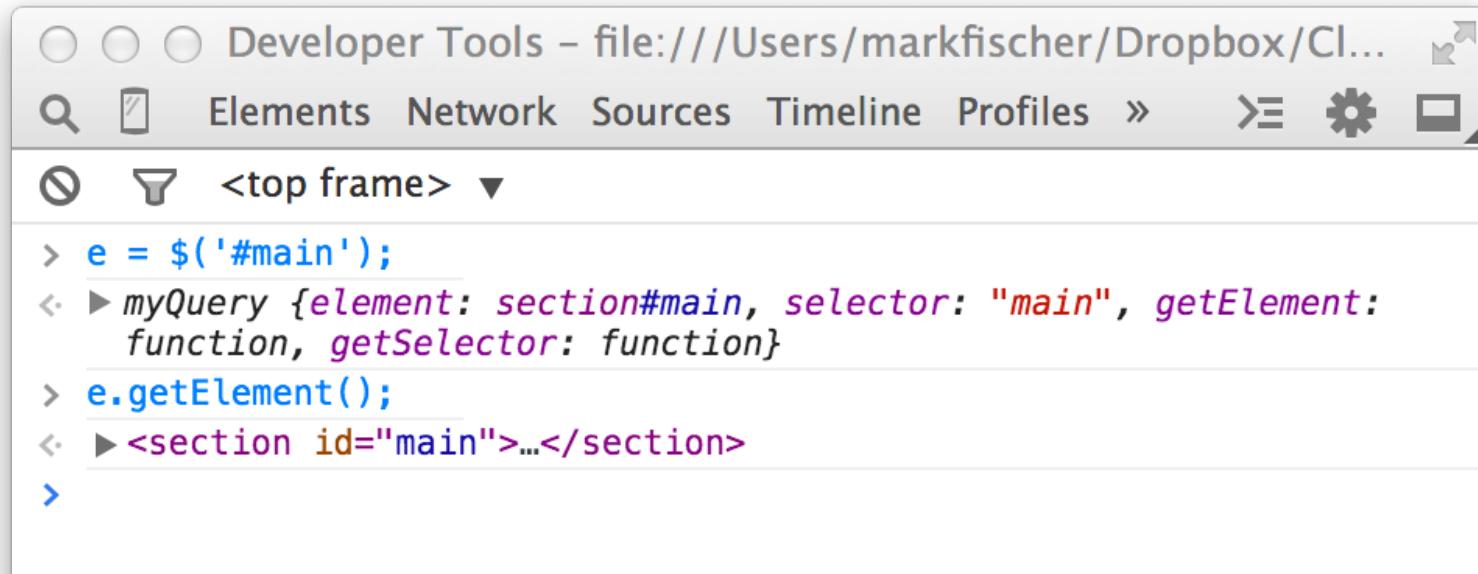
Returning Objects

```
function myQuery(selector) {  
    this.element = null;  
    this.selector = selector;  
  
    // See if selector starts with a #.  
    // If so we're looking for an ID  
    if (selector[0] == '#') {  
        // Strip off the # sign  
        var selector = selector.substring(1, selector.length);  
        var element = document.getElementById(selector);  
  
        myQobj = new myQuery(selector);  
        myQobj.element = element;  
        return myQobj;  
    }  
  
    var $ = myQuery;
```



prototype Methods

```
myQuery.prototype = {
    getElement:      function() {
        return this.element;
    },
    getSelector:     function() {
        return this.selector;
    },
}
```

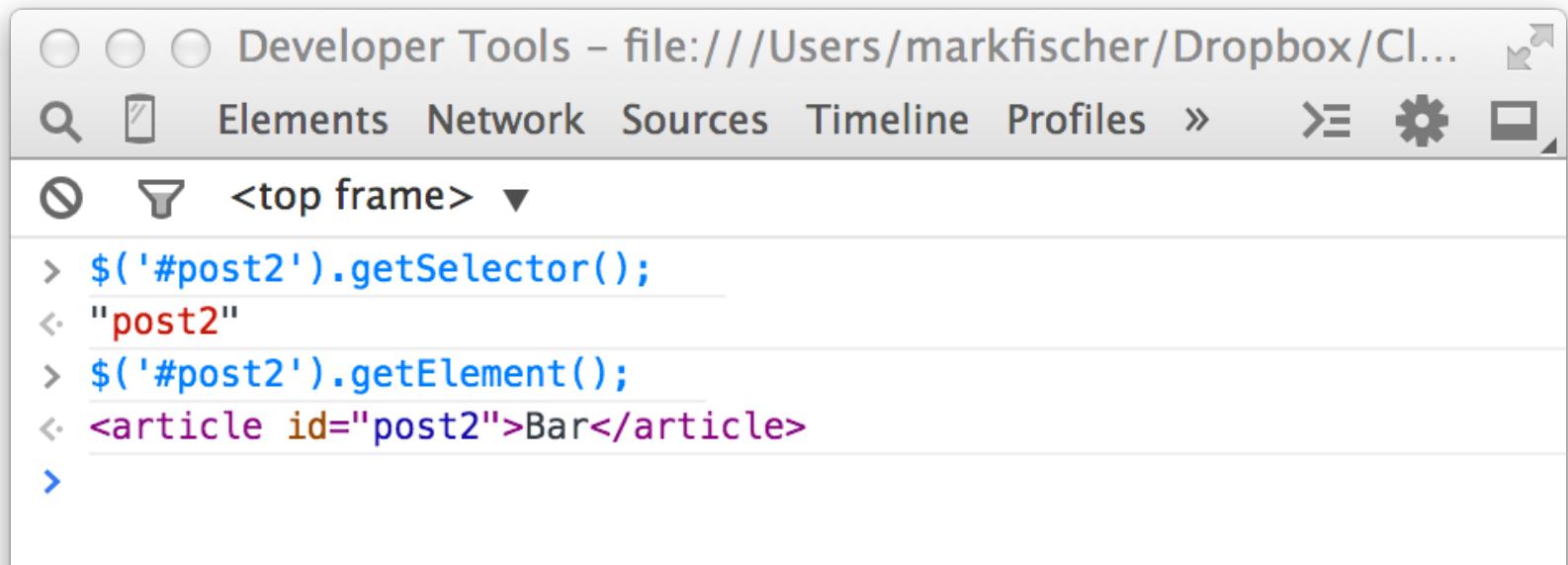


The screenshot shows the Developer Tools console in Google Chrome. The title bar reads "Developer Tools - file:///Users/markfischer/Dropbox/Cl...". The toolbar includes icons for search, elements, network, sources, timeline, profiles, and settings. The main area displays a call stack:

```
<top frame>
> e = $('#main');
<- ► myQuery {element: section#main, selector: "main", getElement:
  function, getSelector: function}
> e.getElement();
<- ► <section id="main">...</section>
>
```

Function Chaining

- Supports function chaining.
- The return value from the function call is an object, which has methods we can call.
- Don't need intermediate variables.



The screenshot shows the Google Chrome Developer Tools interface. The top bar displays 'Developer Tools - file:///Users/markfischer/Dropbox/CI...'. Below the bar, the 'Elements' tab is selected, along with 'Network', 'Sources', 'Timeline', and 'Profiles'. To the right of the tabs are icons for settings and other developer options. The main area shows a DOM tree under the heading '<top frame>'. The tree includes nodes for '#post2' (selected), 'post2', '#post2', and an article element with id='post2' containing the text 'Bar'. Below the tree is a command history panel with the following entries:
> \$('#post2').getSelector();
< "post2"
> \$('#post2').getElement();
< <article id="post2">Bar</article>
>

jQuery

- This is basically what jQuery does.
- More methods and selector types.
- There's a lot more edge cases handled, and checks made.
- jQuery 'plugins' just add their own function calls to the **jQuery prototype** property.

<http://code.jquery.com/jquery-2.2.1.js>

And now for something
moderately different