JavaScript Introduction

Client-side versus server-side programming

- client in our case is the user agent (e.g. the browser)
- server side programming (back-end): write the code that runs on the server. Some common examples: Java, PHP, Python. (Can also have server-side JS -- node.js)
    - Client side of web app makes HTTP requests (GET, POST, etc.) to server, those requests trigger *program execution on the server*, server sends results back to client.
- client-side programming (front-end): write the code the runs in the browser (JS, HTML, CSS). Your client-side programs might make requests to server(s) or not.

Including JS in HTML

- Use `<script>` tags for
    - embedded scripting
    - external JavaScript files
- Script tag can be anywhere in HTML head or in body, but it's usually in head.

External:

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

Embedded:

```
<script>
   var what = "World"
   alert("Hello " + what + "!");
</script>
```

The Document Object Model (DOM)

When we write JS for the web, our code manipulates the HTML documents in our web application, so we need to understand how the browser represents HTML documents internally.

The life-cycle of a web app consists of two phases: *page building* and *event handling*. We'll focus on *page building* for now.

In order to display an HTML document (rendering), a browser constructs a tree (a hierarchy) of *objects* that represent all the HTML elements in the document. In other words, the DOM represents the UI of your web app. Each node in the tree (each element) contains information about the kind of element, its styles, etc. This tree is the *Document Object Model* (DOM).

A page's HTML is the *blueprint* the browser uses to construct the DOM. The DOM and the HTML code might not match exactly: for example, if there are syntax errors in the HTML code, the browser will make some guesses and construct a DOM despite the errors.

We say the DOM is a tree because:

- it has a root node that case no ancestors (parents): the HTML element,
- all nodes have zero or more children (for example, in `<section><p></p><div></div></section>`, the `section` element has two children: `p` and `div`),
- all nodes, except the root, have exactly one parent.

JS code interacts with the DOM to get information (form input, user interaction) and display new information, visual effects, etc.

How the DOM is made

When user enters a URL to request a particular file on the web (www.example.com/index.html):

- Browser requests index.html from www.example.com server and starts processing server's response (downloading index.html)
- index.html is processed top to bottom, element by element
  - Whenever a `link` tag is encountered, like `<link rel="stylesheet" href="style.css">`, browser downloads the resource pointed to by `href`.
  - Whenever a `script` tag is encountered, browser downloads **and runs** the JS code. That code might modify the DOM.
  - In other words every time browser encounters a script or a stylesheet, it pauses to fully process those files before continuing to process any other HTML elements below.
- CSS rules are also processed in the order they are encountered and the style information is combined with the DOM information.
- Finally the DOM is rendered and painted in the browser window. The browser emits an event to indicate that the DOM has loaded and we can write code to listen for that event.
- At this point, more JS code might execute in response to various events like mouse clicks and so on. This code may modify the DOM further.

JS in the browser

- The browser window is the execution environment for your JS code and it provides some objects and functions: `document`, `console`, `alert` and much more.
- Use `.` syntax to access any object's properties and methods: e.g. `document.getElementById("foo")` calls the `getElementById` instance method on `document`
- Use `console.log(whatever)` to print messages to the browser's dev tools JS console
- The *global scope* in a browser is represented by the `window` object: so writing `document.getElementById` is the same as `window.document.getElementById`.
  - We say that `window` is the *global object*.

    - The `window` object represents a window that contains a DOM
- We can manipulate the DOM by calling methods in the `document` object.

Some Notes on Basic Syntax and Semantics

- Here's a JS syntax cheat sheet
- A lot of the syntax is similar to Java, but some major differences
    - You declare variables with `var`
    - One way to declare functions is with `function` (examples below)
    - variable types: `boolean`, `number`, `string`, `undefined`, `null`
        - `undefined` comes up when a variable or property hasn't been initialized yet (or doesn't exist)
    - There are no type annotations when declaring variables or functions (yay! less code to write!) (boo! harder debugging!)
- Comparison with Java:
    - JavaScript uses weak typing (It will automatically convert strings to numbers, or vice versa, without warning you. Surprising behaviour is common.)
    - JavaScript uses dynamic typing (you don't have to declare types in your code, the JS interpreter will figure out the types of variables at run-time)
    - JavaScript is interpreted (no compiler)
    - JavaScript has prototypes not classes (you define an object by just... creating it; we'll learn about this later)

Variables and `var` keyword

- Declare a new variable with `var x = 10`
- Variable *declaration* versus *initialization* versus *assignment*:

```
var a = b * 2; // b hasn't been declared yet, ReferenceError
// declare a variable
// (and it gets automatically initialized with value `undefined`)
var x;
// now assign "hi" to it
x = "hi";
// or you can declare and initialize in one line
var y = 10;
```

variable hoisting

- Even if you declare a variable with `var` all the way at the bottom of a function, you can use it anywhere in the function, which can lead to bugs. It's as if the `var` statement gets automatically moved to the top when you actually run your code.

```
var y = x + 1; // y is NaN because x is undefined;
var x = 10;

//is the same as this
var x;
var y = x + 1;
x = 10;
```

Scope of variables

- Every variable has a scope.
- When you declare a function you declare a new scope.

```
// this line is in global scope, it declare x as a global variable
// which means that window.x is now 100.
var x = 100;
// y is not in scope out here
function print() {
  // both x and y are in scope here
  var y = 2;
  console.log(x);
  console.log(window.x);
}
print();
// log message is: 100 100
```

- Outside of any function we have global scope (the `window` object). A page's global scope stays alive as long as the web page is open in the browser. When you reload the page, its global scope resets. When you close the tab or window, that global scope is killed.
- **Terrible Thing #1**: scope and variable-hoisting weirdness:

```
var x = 100;
function print() {
  if (false) {
    var x = 99;
  }
  console.log(x);
}
print();
// log message is: undefined (what??!)
```

    - because of var hoisting, the above example is the same as:

```
var x = 100;
function print() {
  // x declaration at the top means x gets initialized with `undefined`
  // This hides the x that is in the outer scope (x = 100)
  var x;
  if (false) {
    x = 99;
  }
  // this is using the x in the `print` function scope, not outer scope
  console.log(x);
}
print();
// Log message is: undefined
```

- **Terrible Thing #2**: if you assign to a variable without declaring it with `var`, then it becomes an attribute of the global object, `window`.

```
var x = 100;
function doSomething() {
  // Not using `var` below, so this `alert` refers to `window.alert`
  // even though we are inside the function `doSomething` :(
  // We're overwriting window's alert method with the value 2! :(
  alert = 2;
}
doSomething();
console.log(window.alert)
// Log message is: 2
console.log(alert)
// Log message is: 2
```

Beware of weak typing in JavaScript

```
7 + 7 + 7; // 21
7 + 7 + "7"; // 147
"7" + 7 + 7; // 777
true == !"0" // true
true == 1 // true
true === 1 // false -- notice the triple equals
```

Best Practices to Make Your JavaScript Code Less Buggy

- Always declare variables with `var` so you don't add them to global scope unintentionally.
  - You can/should add `use strict;` to the top of each script you write enable strict mode to make the JS engine enforce this and other best practices. Strict mode is great and easier to debug but it's not available in some old browser versions.
- Make your code easier to read by always declaring all variables at the top of their scope

(since they get hoisted to the top anyway)

- Even though you don't have to write type declarations, always be thinking about data types when writing JS code and avoid implicit type conversions. Your future self will thank your past self for writing clear code that is easier to debug.
  - Always use `===` to test equality.

Built-in objects and functions

- getElementById() * You can modify properties of window and document
- The browser fires a couple of events to indicate that the page is ready:
  - `DOMContentLoaded` fires when initial HTML doc has been parsed
  - `load` fires when all images and styles have finished loading as well
- The structure of an external JavaScript File
  - define an “initialization function” as a listener for `load` or `DOMContentLoaded`, so that it will be called when page is loaded and browser is done building DOM.

```
// initialization function
function doStuff () {
  // ...
}
document.addEventListener("DOMContentLoaded", doStuff);
// or...
window.onload = doStuff;
// or...
window.addEventListener("load", doStuff);
```

- It's best to use addEventListener because it allows you to register many different functions to the same event.
- There are lots of other built-in functions and objects: Math, Date

Lab

- Add JavaScript to an HTML document using the various methods (inline, embedded, external)
- use JavaScript operators, expressions, built-in functions and objects to perform calculations
- Display value of variables inside HTML elements
- Dynamically modify HTML element properties in the web browser window (innerHTML)

Last update: 2017-01-26 00:11:45

Source: _javascript_423/lectures/02_js_intro_overview.md