Introduction To JavaScript
Dynamic Content For The Web

Hello, JavaScript

Probably the most radical difference between web publications and print media is *dynamism*, the ability for web documents to change. Print content is *static*: it doesn't change. Once the ink hits the paper, the printed information is permanently fixed, for better or worse.

In contrast, web pages are most often displayed on the pliable medium of a computer screen, so, naturally, clever computer folks have invented ways to update the contents of a page even after it has been displayed. Dynamic web pages can react to the user or post updates from a remote server. The content and style of a given page can be adjusted, modified, or even replaced, creating the possibility for powerful interactive web applications (and not a few dangers besides).

JavaScript is the standard client-side web scripting language. *Client-side* means that it runs in your browser on your computer (your computer is the "client," as opposed to the "server" like Amazon.com or UWG.edu). JavaScript programs are fetched with the XHTML and CSS of a dynamic page and are executed by your browser in response to events, like mouse clicks, timers, or even when the page is finished loading.

What Can JavaScript Do?

I know, right, you're thinking *this is kinda cool in theory, but...*

Here are some of the basic things you can do in JavaScript:

- Find the length of a string:

  ```
  JS> "There are " + "JavaScript".length + " letters in the word 'JavaScript'"
  ```

  There are 10 letters in the word 'JavaScript'

- Do a mathematical calculation:

  ```
  JS> "3628 * 921 = " + (3628 * 921)
  ```

  3628 * 921 = 3341388

- Work with dates and times:

```
JS> "Today's date is:  " + Date()
```

Today's date is: Mon Jul 27 2009 21:39:23 GMT-0400 (Eastern Daylight Time)

- Generate random numbers:

  ```
  JS> "Here is a random number between 0 and 1:  " + Math.random()
  ```

  Here is a random number between 0 and 1: 0.0715726085910191

Exercise 1: Exploring JavaScript

1. Download the JavaScript intro package from the course web
2. Open the index page in your browser
3. Try reloading the page a few times to see what happens
4. Click the magnifier percentages in the upper right corner
5. Open `index.html` in your text editor. Find the div with ID `basic`. Notice that the XHTML just contains dummy content; the real content is generated and displayed by JavaScript.
6. Open the JavaScript file in your text editor and examine the `basicJavaScript` function to see how the first section of the page is created

Exercise 2: Fixed Dice

The "dice" in that section are just a list of empty squares. Let's fix them up so they show a random number between 1 and 6.

1. Take a look at the JavaScript file in your text editor. The function that is supposed to roll the die is called, incredibly, `rollDie`:

   ```
   function rollDie(numberOfSides) {
       return " ";
   }
   ```

   The function takes one parameter, `numberOfSides`, which allows you to decide how many sides the die has. To get a "normal," 6-sided die, you would say:

   ```
   rollDie(6)
   ```

   If you were playing a role playing game and needed a 20-sided die, you could say:

   ```
   rollDie(20)
   ```

   As you can see, though, the method is incomplete: right now it just returns a string with a single space, which is why the squares are empty.

2. As a first step, let's just have the function return whatever value is passed in for the number of sides. Change the return statement to look like this:

   ```
   return numberOfSides;
   ```

   Save your work, then switch to the browser and open or reload the page. All the dice should be showing 6 (woo-hoo!).

3. JavaScript includes a built in function called `Math.random()` that returns a random number between 0 and 1; you can see it working in the Basic section of the page. How can we scale the number `random()` returns to get a random number between 1 and `numberOfSides`? Addition won't work:

   ```
   numberOfSides <- 6
   numberOfSides + Math.random(): 6.0715726085910191
   numberOfSides + Math.random(): 6.3020728368403248
   ...
   ```

   Right! Not addition, multiplication! We can use multiplication to scale the random number to a value between 0 and `numberOfSides`; change the return statement to:

   ```
   return Math.random() * numberOfSides;
   ```

   Hmm... well, our numbers are random, but they aren't integers! Let's fix that.

4. To make all of our numbers integers we can use the floor function to chop off the decimal:

   ```
   return Math.floor(Math.random() * numberOfSides);
   ```

   Save your work and reload the page in the browser to make sure everything works.

5. There is one problem left for you to fix: you probably noticed that the dice show values from 0 to 5. What can you do to this range so that it shows the numbers from 1 to 6? Implement your solution, using the other code in the script as a guide.

Here is some more information about JavaScript's math functions:

- random: http://www.w3schools.com/jsref/jsref_random.asp
- floor: http://www.w3schools.com/jsref/jsref_floor.asp
- Math object: http://www.w3schools.com/jsref/jsref_obj_math.asp

JavaScript and XHTML, Loops

If you looked at the XHTML for the dice, you probably noticed that it looks a bit odd:

```
<h3>Dice</h3>

<ul id="dice">
    <li name="die"></li>
    <li name="die"></li>
```

```
    <li name="die"></li>
    <li name="die"></li>
    <li name="die"></li>
</ul>
```

Weird, huh? *How does a bullet list end up being displayed with no bullets in one row? How does a list of identical, empty items get filled with different content? What about the bunnies?*

Easy, easy, keep your shirt on. Let's take a look at the answers:

- *How does a bullet list end up being displayed with no bullets in one row?*

  Easy: CSS:

  ```
  #dice li {
      display: inline;
      list-style-type: none;
      ...
  ```

  We used this same trick for the magnifier control; it is a fairly common web implementation *idiom*, or pattern. You can read more about the CSS at w3schools:

  http://www.w3schools.com/css/css_list.asp

- *How does a list of identical, empty items get filled with different content?*

  The answer to this question is so big we need a new section just to hold it... First, though, a thought experiment: what do you think will happen if we add additional elements with `name="die"`?

DOM: The Document Object Model

The Document Object Model (DOM) defines a standard way for JavaScript programs to access and manipulate XHTML documents. The DOM is an object-oriented, tree-structured representation of a document. It is object-oriented because every component of the document is represented by an object:

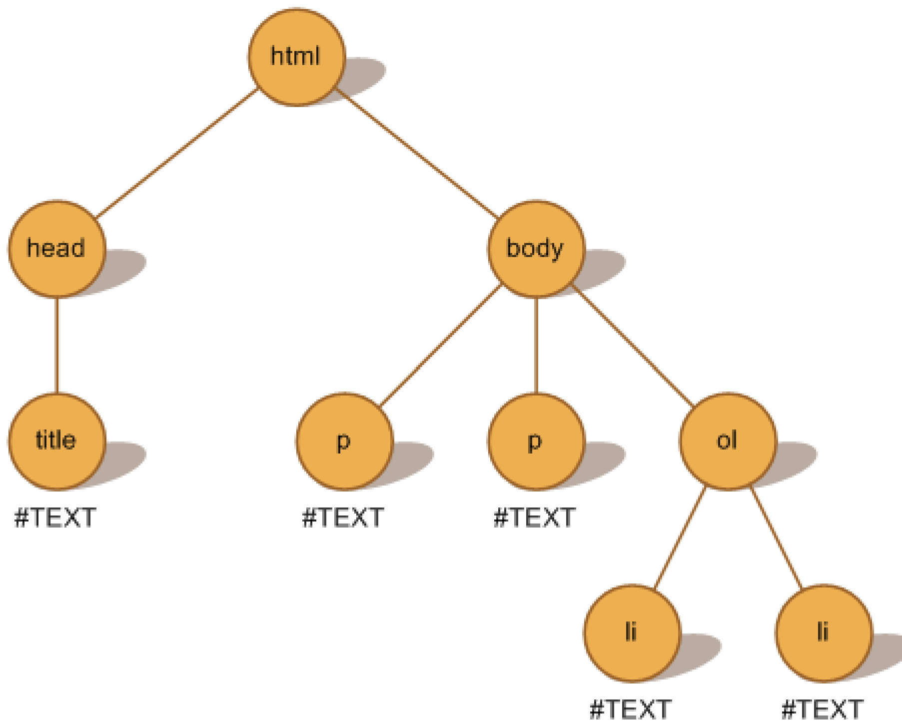
- element objects like `<body>` and `<p>`
- attribute objects like `id` and `href`
- text objects

The DOM is also a tree-structured: the objects, called "tree nodes," are organized into a hierarchy of parents and children, ancestors and descendants. Containing elements are ancestors of contained elements; for example, the paragraphs, lists, and text that make up a page are all descendants of the `<body>` element. Child elements are direct descendants of their parent element, e.g. the items in a list are children of the list element—`<ul>` or `<ol>`—that contains them.

Let's take a look at an example. This XHTML fragment:

```
<html>
    <head>
        <title>...</title>
    </head>
    <body>
        <p>...</p>
        <p>...</p>
        <ol>
            <li>...</li>
            <li>...</li>
        </ol>
    </body>
</html>
```

corresponds to the following DOM tree visualization:



Accessing Elements

There are several ways to access tree elements from JavaScript,:

- retrieve a single element by its CSS id:

  ```
  var basic = document.getElementById("basic");
  ```

- retrieve all the elements with the same tag name:

  ```
  var list_items = document.getElementsByTagName("li");
  ```

- retrieve all the elements with the same name attribute:

```
var dice = document.getElementsByName("die");
```

Since the DOM is a hierarchical structure, you can also access descendants of other elements, restricting your search to part of the document (a "subtree") rather than the entire document. For example, the text magnifier control retrieves the `<li>` elements that are descended from the magnifier container (`id="resize_text_sizes"`) without affecting the other `<li>` elements in the document (note the use of "`magnifier`" instead of "`document`" in the second line):

```
var magnifier = document.getElementById("resize_text_sizes");
var sizes = magnifier.getElementsByTagName("li");
```

Exercise 3: More Dice

OK, now that we've covered the basics, let's try an *actual* experiment:

1. Open `index.html` in your favorite text editor
2. After the list of dice, add a paragraph to say what you're going to do:

```
    ...
    <li name="die"></li>
</ul>

<p>Another list of dice:</p>
```

3. Make a copy of the list of dice below the paragraph you just added
4. Save your work, then reload the page in the browser

The JavaScript function that makes all this work is called `rollDice()`:

```
function rollDice() {
    var dice = document.getElementsByName("die");

    for (var i = 0; i < dice.length; i++) {
        dice[i].innerHTML = rollDie(6);
    }
}
```

- The first line gets all the elements named "die" (i.e. elements with `name="die"`) and puts them in a list variable we've cleverly called "dice."
- Then we use a for-loop to run some code for each die in the list
  - the loop sets the variable `i` to the number of each die in the list in turn
  - `dice[i]` is the actual list item from the XHTML

– `innerHTML` is the content inside the list item
– `rollDie(6)` rolls a six-sided die (this calls the function we worked on earlier)

To recap: this statement:

```
dice[i].innerHTML = rollDie(6);
```

says, effectively, “replace the value of the current die from the list of dice with the result of rolling a six-sided die.”

for loops and the `i` variable

Normally, single-letter variable names are frowned upon as poor development style. Since single letters convey ambiguous meaning, at best, they do not support a low representational gap between the name used and the concept represented. Single-letter variable names are a sure sign of code that is hard to understand and maintain.

One of the rare exceptions to the “no single-letter variable names” is the use of the variable name `i` as the index variable in a for-loop. This pattern is ubiquitous and longstanding: variations of the construct

```
for (i = 0; i < ...; i++) {
    ...
```

have been in use for so long in so many different languages that purpose of the variable `i` is instantly understood by developers.

Can you think of any other exceptions where single-letter variable names might be acceptable?

Writing Your Own Functions

You’ve seen several functions throughout this tutorial while you’ve been reviewing the code and making changes. However, we haven’t really explained what functions are. A *function* in JavaScript is a reusable block of code with a descriptive name. JavaScript functions are very similar to methods in languages like C# and Ruby.

function: a reusable block of code with a descriptive name

Just like the methods and functions in other languages, functions in JavaScript can take *parameters*, which provide extra information that the function needs to do its job. For example, this function:

```
function rollDie(numberOfSides) {
    ...
```

has one parameter, `numberOfSides`, because the function needs to know what kind of die you want to roll: 6-sided, 20-sided, or something else. Of course, not all functions need parameters, e.g. `basicJavaScript()`.

JavaScript functions can also return values. The `rollDie(...)` function is a good example: it returns the result of the die roll. Return values can be used just like any other value in a JavaScript program.

Defining Functions

Defining functions in JavaScript is straightforward:

```
function name(parameters) {
    // code goes here
}
```

The four parts of a JavaScript function are:

- the keyword `function`, which lets JavaScript know you want to define a new function
- the function's name, which should provide a summary description of what the function does
- an optional list of parameters
- the code that makes up the function surrounded by curly braces

Exercise 4: Dice Total

1. Open the JavaScript file `script.js` in your favorite editor
2. Find the function description for calculating the dice total. You'll see that the function has not been implemented; there is a placeholder comment indicating that you need to implement the function.
3. Replace the comment with the function declaration:

   ```
   function calculate_dice_total() {
   }
   ```

4. For now, let's have the function just return 0; we'll come back and fill in the details later:

   ```
   function calculate_dice_total() {
       return 0;
   }
   ```

   This practice is known as *incremental development* and is an important skill for any software developer to master. You should work in small chunks, with the goal of getting some small piece of the problem implemented and tested before moving onto the next chunk.

   But it doesn't work, you're thinking. That's OK! The function has enough functionality that we can test displaying the dummy total; we'll come back and fix up the calculation later.

5. Find the function `display_dice_total()`, which is supposed to show the total of the dice in the basic section of the page. Using the function `basicJavaScript()` as a guide, implement the method as follows:

   ```
   var basic = document.getElementById("basic");
   basic.innerHTML += "<p>The sum of all the dice below is: " + -1 + "</p>";
   ```

   This code appends the paragraph `"<p>The sum..."` at the end of the basic section.

6. Save your work, then reload the page in your browser to make sure what you have so far is working. If everything is correct, you should see this line at the end of the basic section:

   The sum of all the dice below is: -1

7. Now, replace the dummy value "-1" with a call to the function `calculate_dice_total()`. Save your work and reload the page in the browser; you should now see this line at the end of the basic section:

   The sum of all the dice below is: 0

   This lets you know that the call to the calculate function is working correctly.

8. The last thing you need to do is fix up the function `calculate_dice_total()`. Use the function `rollDice()` as a model.

   a. First, create a variable to hold the dice and get them from the page:

   ```
   var dice = document.getElementsByName("die");
   ```

   b. Next, we need a variable to hold the total. Set its initial value to 0:

   ```
   var total = 0
   ```

   c. We need to iterate over each die in the list. Fortunately, we can use the same code from `rollDice()`:

   ```
   for (var i = 0; i < dice.length; i++) {

   }
   ```

   d. Inside the for-loop, we need to 1) get the value of the die (`dice[i].innerHTML`), 2) convert the value to an integer number with `parseInt`, and 3) add the result to the total:

   ```
   total += parseInt(dice[i].innerHTML);
   ```

   e. Finally, after the for-loop's closing brace (}), return the total:

   ```
   return total;
   ```

9. Save your work, then reload the page to make sure everything works correctly.

Events, Handlers, and Binding

JavaScript code executes in response to *events*, which we will helpfully define for you:

> **event:** something that happens

Clears everything right up, doesn't it?

A few of the somethings that can happen include:

- mouse movement and clicks
- timer expiration

- submitting a form

JavaScript responds to events using *event handlers*, which are normal functions that get called when a particular event happens. One of the most common JavaScript events is `window.onload`, which fires when the page is finished loading. For example, the page you have been working with throughout this module uses the load event to populate the basic section and set up the rest of the page:

```
// -----------------------------------------------------------
// JavaScript initialization

function initialize() {
    bindTextMagnifier();
    bind_font_selector();

    basicJavaScript();

    rollDice();

    display_dice_total();
}

// call the initialize function when the page is loaded
window.onload = initialize;
```

The last line of this code *binds* the function `initialize` to the `window.onload` the event, so `initialize` is now the event handler for that event and will be called whenever the event fires.

Text Magnifier

Let's take a closer look at how this process works for the text magnification control:

```
/**
 * Bind the event handlers for the text magnifier click events.
 */
function bindTextMagnifier() {
    var magnifier = document.getElementById("resize_text_sizes");
    var sizes = magnifier.getElementsByTagName("li");

    // Bind each button's onclick event to magnifyText
    for (var i = 0; i < sizes.length; i++) {
        sizes[i].onclick = function() {
            return magnifyText(this);
        }
    }
}
```

We've already examined the first two lines of the function; they retrieve all the `<li>` elements under the text magnification container `resize_text_sizes`:

```
var magnifier = document.getElementById("resize_text_sizes");
var sizes = magnifier.getElementsByTagName("li");
```

The for-loop should also be familiar by now; it states, effectively, "for each item in the `size` list, do this code":

```
for (var i = 0; i < sizes.length; i++) {
```

The actual event binding happens inside the body of the loop:

```
sizes[i].onclick = function() {
    return magnifyText(this);
}
```

- `sizes[i]` is the current list item, so `sizes[i].onclick = ...` sets the current list item's onclick handler to the function on the right side of the `=`
- `function()` defines a new, *anonymous* function as the handler for the click event. As the name suggests, an anonymous function is a function without a name; they are commonly used for event handlers since the function only ever needs to be called via the onclick event and not elsewhere in the JavaScript program. If this doesn't make sense to you right now, don't worry: you'll get more comfortable with anonymous functions as you gain experience with JavaScript.
- the body of the event handler function consists of the single return statement:

  ```
  return magnifyText(this);
  ```

  "`this`" is a special JavaScript keyword that means "the current object," or, in this context, "the item that was clicked on." If you clicked the "75%" item, "`this`" would point to that element.

  When you click one of the magnifier items, it's event handler calls the function `magnifyText`, passing itself (the item you clicked on) as the parameter to `magnifyText`. `magnifyText` uses the text of the clcked item—e.g. "125%"—to set the new text size of the content of the document.

Exercise 5: More Magnification

- Add three new list items to the magnification control in `index.html`: 50%, 150%, and 200%. Be sure to add them in the correct order: your users will expect it.
- Reload the page in your browser and try out the new controls. Do they work as expected?

Usability And Unobtrusive JavaScript

At the start of this module, I said that JavaScript brought both potential and danger to web development. You've seen a small glimpse of the potential; can you guess what the danger is?

Have you ever seen a really bad CG effect in a movie? The camera pans dramatically, when suddenly your senses are jarred by a creature, explosion, or alien warship that looks like it was cut out of a magazine and pasted to your TV. As slick as the effect may be, it doesn't blend with the scene. It is "loud;" it stands out and calls attention to itself, rather than cooperating harmoniously with the actors, sets, and other elements of the film. As a result, both the scene and the effect are spoiled.

For most of the life of the Web, JavaScript has been like one horribly long and especially egregious* CG effect. Confusing, overly-complicated navigation systems, flashy-blinky trinkets, and obnoxious advertising that obscured the actual content of the site are but a few examples of JavaScript's sordid history. Examples abound; in fact, most of the JavaScript samples you'll find on the web are obtrusive. Signs of obtrusive JavaScript include:

- sites that don't work or have crippled functionality without JavaScript
- sites that use browser-specific JavaScript
- sites that freely intermix JavaScript with HTML
- sites that value JavaScript gizmos over usability

Exercise 6: Obtrusive JavaScript

- Find 3 examples of sites that use obtrusive JavaScript. What makes it obtrusive? Why is the obtrusive behavior bad? What would you do to fix it?

Toward Unobtrusiveness (Unobtrusivity?)

Fortunately, the recent advent of serious JavaScript applications has kindled thoughtful reflection about the proper role of JavaScript on a modern site. While there are no hard-and-fast rules, some simple guidelines can help ensure the sites you design and implement don't exhibit obtrusive behavior:

- **Put usability first!** Nothing should trump usability on a site, so if a JavaScript effect or technique hurts the user experience, get rid of it.
- Apply **Separation of Concerns**; put
  - structure and content in XHTML
  - style in CSS
  - behavior in JavaScript
- As a corollary to the above, **keep JavaScript code out of the XHTML!** Many sites intermix JavaScript with their markup, leading to inconsistent behavior, bugs, and poor maintainability. Modern JavaScript techniques and browser support eliminate the need and justification for mixed behavior and markup.

* Don't believe me? Do *you* remember pop-up ads?

- Make sure your site **degrades gracefully**. There are many legitimate reasons a user might not have JavaScript enabled, including:
  - firewall settings outside of their control
  - limited browser capabilities (e.g. mobile or imbedded browsers)
  - security concerns

  Always test to ensure that essential elements of site functionality, like navigation, work with JavaScript disabled.

Exercise 7: Tying It All Together

Practice what you've learned in this module by implementing the last piece of missing functionality on our example JavaScript page: font selection.

- the font names at the bottom of the page should be displayed in the appropriate font
- clicking a font name should change the Basic section to clicked font

The functions have already been started for you. Use the descriptive comments in the code and the functions you've already worked on to help you. Don't forget to work incrementally!

Additional Resources

- w3schools JavaScript tutorial: http://www.w3schools.com/js/default.asp
- *DHTML Utopia* (book with free PDF excerpt): http://www.sitepoint.com/books/dhtml1/
- Unobtrusive JavaScript: http://en.wikipedia.org/wiki/Unobtrusive_JavaScript
- CSS style names and their JavaScript counterparts: http://codepunk.hardwar.org.uk/css2js.htm