Section Notes

Contents

1 JavaScript

JavaScript is a scripting language that developed alongside HTML during the browser wars between Microsoft Internet Explorer and Netscape Navigator. Though its name implies some relationship to Java, the Java in JavaScript reflects only the business relationship between Sun and Netscape; Sun owned the trademark

while Netscape did most of the development on the language. It is different from the other web-related programming languages you've seen so far in that it is run *client-side*, which means all code is executed on the local machine of the user, whereas PHP is processed by the web server. This has a few important implications. First, whereas users cannot disable your PHP pages to alter their functionality, most browsers in use today have many options to restrict what type of JavaScript can be executed. Scripts that create popup windows containing ads without asking the user are commonly blocked in this way. Furthermore, since JavaScript is interpreted by the user's web browser, the source code must be fully available to the browser and by extension to the user. It's therefore extremely unsafe to hardcode any sensitive information into a piece of JavaScript.

The other important property of JavaScript is that it is interpreted, not compiled. In other words, you don't compile a piece of JavaScript any more than you'd compile a PHP page. Whatever web browser is accessing a site containing JavaScript will dynamically execute the code according to its own interpretation of the JavaScript standard. Since JavaScript is designed to fail gracefully, you may write (and probably have visited many sites with) JavaScript that contains errors without ever knowing it—unlike even PHP, your page will not fail to render if it has a JavaScript syntax error. Instead, the part of the site dependent on JavaScript will simply stop working, or even more insidiously, it may continue to work in an unexpected way with warnings flagged by the interpreter. A debugging tool like Firebug is critical for writing clean, predictable JavaScript.

1.1 Syntax

1.1.1 Variables

JavaScript's syntax is very similar to C's, though it also bears resemblance to PHP in many aspects. Variables in JavaScript are declared by assigning to them:

```
age = 25;
```

This creates a global variable named age that will be set to 25. Note the lack of a type specifier; like PHP, JavaScript is loosely typed, meaning you don't need to tell it explicitly that any given variable is an `int` or a `string`.

The variable created in the above example will be global, just like a variable declared outside of any function in C. This is not always what we want; a list iterator, for example, should not be visible outside the function using it. JavaScript therefore provides the `var` keyword.

```
var i = 0;
```

This declares a local variable named `i`. Even if there is already a global variable named `i`, any code in the same function following this declaration will use the local version. In general, you should declare most variables local unless they need to be global; this will help you avoid running into name conflicts later.

1.1.2 Arrays

A JavaScript array may be declared by setting a variable equal to some expression in brackets.

```
var array = [];
var oneThroughFive = [ 1, 2, 3, 4, 5 ];
```

Arrays in JavaScript are dynamically resized as necessary, so there's no need to specify any size. If you do try to reference an "out of bounds" index, such as one you haven't initialized yet, its type will be considered `undefined`. JavaScript arrays can contain any type of value, even heterogeneously within the same array, so

```
var array = [ 1, "two", function() { return 3 }, 4.001 ];
```

declares a valid array in JavaScript.

1.1.3 Operators

JavaScript supports essentially the same operators as C. However, there are a few important differences.

- `+`
  The `+` operator performs addition as you'd expect, but it can also concatenate strings like `.` in Perl or PHP.

  ```
  string1 = "See";
  string2 = "Spot";
  string3 = "run";
  document.write(string1 + " " + string2 + " " + string3);
  ```

  This code snippet prints "See Spot run" to the page.

- `==`
  JavaScript has the same `=`/`==` semantics as C, but `==` can also be used to test equivalence of strings. Since JavaScript is loosely typed, `==` will attempt to perform an appropriate conversion so that, for example,

  ```
  "1" == 1
  ```

  evaluates to `true`. It also provides `===` for comparing both the value and the type of a variable. `"1" === 1` will evaluate to `false`.

- `typeof`
  This operator returns the type of its operand, which will be `number`, `boolean`, `string`, `object`, `function`, `null`, or `undefined`. In JavaScript, `null` means "having no particular value" and is the type of the empty object, whereas `undefined` is the value of any variable or array index you haven't initialized yet. So this code

  ```
  dogName = "Spot";
  document.write(typeof dogName);
  ```

  will write "string" to the page.

1.1.4 Statements

Like C, JavaScript provides the if-else conditional statement as well as for, while, and do-while loops. JavaScript also has a switch statement, but it is more versatile than C's. Whereas C's case labels can contain only integer and character literals, JavaScript also supports floating-point and string literals in case labels. So

```
switch (strangeVar)
{
  case 5:
    document.write("5");
    break;
  case 3.14:
    document.write("pi");
    break;
  case "hello":
    document.write("greeting");
    break;
```

```
  default:
    document.write("unknown");
    break;
}
```

is valid JavaScript. Case labels still may contain only literals; you cannot use another variable or a conditional expression like `x < 26` within one.

- `for...in`
  JavaScript provides a statement similar to PHP's foreach to iterate through an array or object with many elements or properties.

  ```
  var weekArray = ['Monday', 'Tuesday', 'Wednesday'];
  for (var day in weekArray)
      document.write(day + " ");
  ```

  This code snippet will print "Monday Tuesday Wednesday ".

1.1.5 Functions

JavaScript really starts to get interesting when you consider functions. JavaScript functions are first-class objects, which mean they have the same status as any variable; they can be created dynamically and passed to or returned from other functions.

This comes most in handy when defining what are known as event handlers. JavaScript lets you associate functions with specific actions taken by the user or the user's browser, which are called events.[1]

```
<input type="text" onfocus="alert('Enter your name');"/>
```

would create a textbox that pops up instructions when you give it focus (move the cursor inside it). Likewise, you could define some function

```
function alertName()
{
    alert("Enter your name");
}
```

and then have the input call that:

```
<input type="text" onfocus="alertName();"/>
```

Event handlers can see additional information about the circumstances under which they were called. This information is contained in the optional event parameter passed to event handlers.

```
function alertName(event)
{
    var triggeringObject = event.srcElement;
    alert("You clicked on the " + triggeringObject.value + " button");
}
...
<input type="button" value="Click Me" onclick="alertName(event);"/>
<input type="button" value="Don't Click Me" onclick="alertName(event);"/>
```

This script would enable you to see which button was clicked even though both call the same event handler.

[1] For a full list of events, see http://www.w3.org/TR/DOM-Level-3-Events/events.html#Events-EventTypes-complete.

1.1.6 Objects

You can think of an object in JavaScript as a container, sort of like a `struct`. Objects in JavaScript can contain both variables, which are usually called properties, and functions, which are then called methods. Creating an object is as simple as including the line

```
car = new Object();
```

You then have an object called `car` that you can modify however you want.

```
car.color = "red";
car["year"] = 2005;
car.makeSound = vroom;
```

That last line would enable you to call `car.makeSound()` and have the function `vroom()` execute, assuming it was defined at some point already. Notice the two different syntax options for accessing members of an object. It's often useful to be able to create a new object with certain properties already filled out on demand. A function called a constructor, written like this:

```
function student(name, class, id)
{
    this.name = name;
    this.class = class;
    this.id = id;
}
```

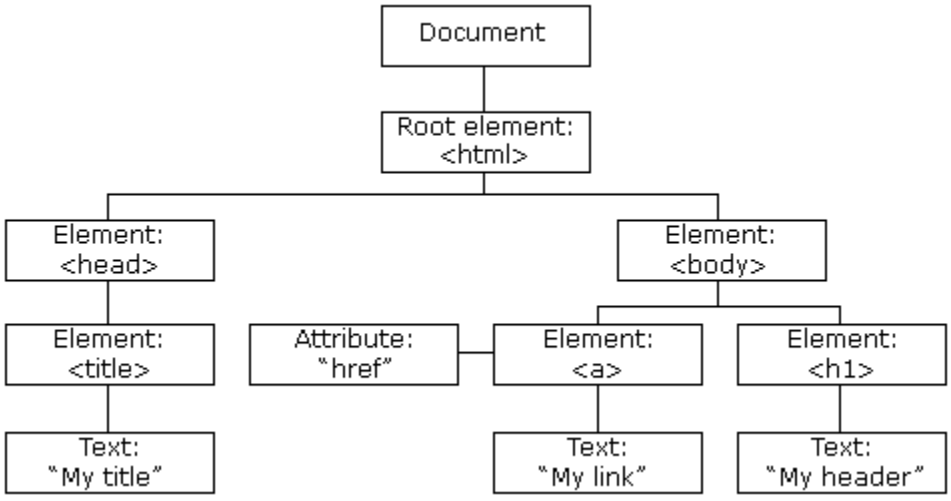
would create a new object with properties `name`, `class`, and `id` corresponding to the ones passed to the function. A new student could then be created with

```
johnHarvard = new student("John Harvard", 2011, 12345678);
```

You'll see many objects created this way when we talk about AJAX and Google Maps.

1.2 The DOM

Objects in JavaScript can contain anything, including other objects. One particularly important hierarchy of objects like this is the Document Object Model. The DOM defines a standard way of treating the elements that make up an HTML page as objects in JavaScript, enabling us to manipulate them programmatically. Here's a simplistic view of the HTML DOM.[2]



[2]http://www.w3schools.com/HTMLDOM/htmltree.gif

Most DOM objects define a similar group of properties and methods with which it's useful to be familiar.

1.2.1 DOM Properties

- `innerHTML`
  The inner text value of some HTML element. For example, the `innerHTML` of the `<i>` tag in `<i>hello <b>world</b></i>` is `hello <b>world</b>`.

- `nodeName`
  The name of an element or attribute. The `nodeName` of the tag `<img src="helloworld.jpg"/>` is `img`, and the `nodeName` of its `src` attribute is `src`.

- `nodeValue`
  The value of an attribute. The `nodeValue` of the `src` attribute above is the string `helloworld.jpg`.

- `parentNode`
  A reference to a node's parent.

- `childNodes`
  An array containing a node's children.

- `attributes`
  An array containing the attributes of a node.

- `style`
  A special object representing the style of a node. The style object itself has properties representing the various CSS style attributes an element can possess. For example, `[node].style.backgroundColor = "red"` is equivalent to styling the node with `[node] { background-color: red; }` in CSS.

1.2.2 DOM Methods

- `getElementById(id)`
  Gets the element with a given ID (specified with the `id` attribute) under this node.

- `getElementsByTagName(name)`
  Get all elements under this node with the given tag name.

- `appendChild(node)`
  Add the given node to the children of this node.

- `removeChild(node)`
  Remove the given node from the children of this node.

Our root for accessing DOM nodes is usually the document object, which is globally accessible from any function on a page. The document object represents the entire page and everything contained within it. Therefore, to find the DIV with the ID `purpleDiv` and make its background purple, we could do something like this:

```
document.getElementById("purpleDiv").style.backgroundColor = "purple";
```

When this power to manipulate the document is combined with JavaScript events and handlers, a page can become incredibly dynamic in itself without ever asking the server for information.

```
<html>
    <script type="text/javascript">
        function turnPurple()
        {
            document.getElementById("purpleDiv").style.backgroundColor = "purple";
        }
    </script>
    <body>
        <div id="purpleDiv">Turn me purple!</div>
        <input type="button" value="Do it" onclick="turnPurple();"/>
    </body>
</html>
```

This page, for example, implements a button that can turn a neighboring DIV purple with a single click. But this example doesn't do justice to the vast changes that one can effect in a page through the DOM. A page can be entirely reconstructed from the bottom up by adding and removing nodes with the desired content.

2 AJAX

One of the problems web applications have traditionally faced is the issue of dynamically changing content. HTTP is a connectionless protocol, meaning that you do not maintain a continuous connection to a website when you visit it. In order to allocate bandwidth and server resources most efficiently, HTTP was designed to allow clients to connect, download enough data for one page, and then disconnect until the next request. This paradigm makes many desirable effects impossible. For example, it would be impossible to implement an effective chat room via HTTP because no one could see what anyone else was saying without hitting refresh; as such, we have a plethora of web-based bulletin boards but almost no web-based chat rooms (and those that exist are usually Java applets, bypassing HTTP entirely).

AJAX, which stands for Asynchronous JavaScript and XML, solves this problem by enabling client-side scripts to make HTTP queries independent of the page in which they are contained (hence the "asynchronous" part). The advent of AJAX precipitated large shifts in what functionality a web site could implement without severely hurting the performance of its host or inconveniencing the user past the point where the functionality would be worth it. Before AJAX, Google's built-in autocomplete, for example, would have required Google to transmit enormous amounts of data in response to every request for its homepage (so the autocompletion could be calculated clientside) or the user to click a submission button every time he wanted suggestions. AJAX allows this feature to be implemented with extremely low overhead (all of index.html need not be transmitted every time the user adds a new character to his search term, nor must Google provide its dictionary to the browser) and total transparency to the user (no button to click).

At the core of any AJAX query is the XMLHttpRequest object. This object provides functionality to send data to a server, await a response in one of two forms, and take action when a response is received.

2.1 The XMLHttpRequest Class

2.1.1 Attributes

- `readyState`
  A code that represents the status of the XMLHttpRequest. Possible values include the following:
  0 - not initialized
  1 - connection established
  2 - request received
  3 - answer in progress
  4 - done

- `status`
  The HTTP status of the request. HTTP statuses are things like 404 for page not found, 403 for access denied, and 200 for request OK.

- `responseText`
  One of two properties that can hold data returned from a request. `responseText` simply holds the response as a string of characters.

- `responseXml`
  The other property that holds returned data. `responseXml` can contain structured XML data which can then be accessed via DOM just like the elements of an HTML page.

- `onreadystatechange`
  The function to call when the `readystatechange` event is dispatched. In other words, this function will be called when a request goes from "in progress" to "done" (or between any other pair of states, but this is the one we really care about).

2.1.2 Methods

- `open(mode, url, async)`
  This function opens a connection to a server in preparation for making a request. The `mode` parameter specifies whether parameters of the request should be submitted via method `GET` (in the query string) or method `POST` (encoded in the HTTP request itself). The `url` is the location of the file to request. The final boolean `async` parameter specifies whether the request is to be asynchronous with respect to the browser. That is, a value of `true` executes a standard asynchronous request that will not halt execution of any other scripts on the page. A value of `false` will force the browser to wait for a response before it does anything else. The vast majority of the time, this should be `true`.

  Note that `open` can optionally take `username` and `password` as extra parameters if the page being requested requires authentication of some kind.

- `send(string)`
  This function sends data for a request. If you are using method `GET`, the parameters sent will be part of the query string at the end of the `url` argument to `open()`, so this should always be called as `send(null)`. If you are using method `POST`, key/value pairs can be sent with a call like `send("param1=hello&param2=world")`.

That might seem like a lot to absorb when presented in such a disjointed way, so let's dive into some examples of how AJAX might be used.

2.2 Validating a Password

If you've signed up for an account with Google (GMail, perhaps?), you've probably seen the nifty password validation widget they use to tell you how strong your proposed password is. Whereas most sites requiring registration require you to go through the annoying process of typing a username and password and clicking submit only to find that the username is taken or the password is invalid, often deleting half of your entries in other fields in the process, this system allows you to square away everything in the form the first time. But how?

Let's start by writing our JavaScript.

```
<script type="text/javascript">
// <![CDATA[

    // an XMLHttpRequest
    var xhr = null;
```

The first thing we do is to declare our `XMLHttpRequest` object. It starts off `null`, since we don't need it for anything until it comes time to actually validate a password.

```
/*
 * void
 * requestPasswordCheck()
 *
 * Sends an AJAX request for a password check.
 */
function requestPasswordCheck()
{
    // instantiate XMLHttpRequest object
    try
    {
        xhr = new XMLHttpRequest();
    }
    catch (e)
    {
        xhr = new ActiveXObject("Microsoft.XMLHTTP");
    }

    // handle old browsers
    if (xhr == null)
    {
        alert("Ajax not supported by your browser!");
        return;
    }

    // construct URL
    var url = "checkPass.php?";
    url += "password=" + document.getElementById("password").value;
    url += "&password2=" + document.getElementById("password2").value;

    // get quote
    xhr.onreadystatechange = handler;
    xhr.open("GET", url, true);
    xhr.send(null);
}
```

Next comes the `requestPasswordCheck()` function. Take a look at that first pair of blocks, beginning with `try` and `catch`. This is something you haven't seen before because C does not support it, but most modern programming languages do. More recent languages have concepts of error handling and *exceptions*, messages that are sent to your program when something goes wrong. This allows us to handle something like dereferencing a null pointer more elegantly than crashing the program and leaving an unsightly core dump in its working directory. The `try-catch` block here basically says "attempt to create an `XMLHttpRequest`; if something went wrong (e.g., we were using IE), try to create a `Microsoft.XMLHTTP` ActiveX object instead." If that doesn't work either, we have no choice but to give up. We then construct the URL to which we plan to make our request. Since we're using method `GET`, the URL consists of the actual location of the page we want to access followed by a query string within which we'll pass our prospective password as data.[3] The

[3]Note that this is extremely insecure! It's never a good idea to transmit a password over the internet as plaintext lest someone be listening. Method `POST` wouldn't work either; it moves the password out of the querystring into the body of the HTTP request, but it would still be plaintext. In the real world, an operation like this would be implemented over SSL.

next few lines are the most critical. We set the `onreadystatechange` attribute to `handler` (no parentheses!) so that our object knows to call `handler()` when the request is completed. We then open the connection to the server and make the request.

```
/*
 * void
 * handler()
 *
 * Handles the Ajax response.
 */
function handler()
{
    // only handle loaded requests
    if (xhr && xhr.readyState == 4)
    {
        if (xhr.status == 200)
        {
            var response = xhr.responseXML;
            var resultDiv = document.getElementById("result");
            if (response.getElementsByTagName("response")[0].attributes
                    .getNamedItem("valid").nodeValue == "false")
                resultDiv.style.color = "red";
            else
                resultDiv.style.color = "green";
            resultDiv.style.fontWeight = "bold";
            resultDiv.innerHTML = response.getElementsByTagName("response")[0]
                                        .childNodes[0].nodeValue;
        }
    }
}
```

Let's move on to our handler. When the request experiences any change of state, including the moment at which it's made, `handler()` will be called. Since our `xhr` object is global, `handler()` has full access to all its properties and methods. We're not interested in doing anything until the request is complete (`status` = 4), so that's the only time code is actually run by this function.[4] If the request succeeded, i.e., the HTTP status returned was 200 OK, we make use of the data provided to inform the user of their password's strength. Otherwise we'll refrain from saying anything at all—the password will just have to be validated on submission. This is a case where displaying "AJAX error!" would probably not be terribly useful for the user.

```
/*
 * void
 * checkPassword()
 *
 * Limits number of AJAX requests made per second.
 */
function checkPassword() {
    document.getElementById("result").innerHTML = "";
    var interval = 1000; // ms
    var lastKeypress = new Date().getTime();
```

[4]You can conceive of some reason you might want to inform the user that an AJAX query was in progress—if you knew the query was likely to be very slow, for example.

```
            setTimeout(function() {
                var currentTime = new Date().getTime();
                if (currentTime - lastKeypress > interval)
                    requestPasswordCheck();
            }, interval);
        }
    // ]]>
    </script>
```

Last we have the function that links the AJAX request being made to an action of the user's. We could have a big "Validate" button next to the password fields, but that would be ugly, and it would require the user to participate. We could just set the `requestPasswordCheck()` function as the handler for the `onkeyup` event of the password field, but then every single keystroke would generate a request to the server. Not a good idea at all. We could also set `requestPasswordCheck()` as the handler for the `onblur` event instead, which is like the opposite of the `onfocus` event and would fire when the user selected another field, but what if the user entered his password last and never switched focus away from the password field?

So we set up a neat little function[5] that checks how long it's been since the user last struck a key, actually performing a check only if that interval is over 1000 ms.[6] The function passed to `setTimeout()` is called a *closure*, a function that can remember aspects of the circumstances under which it was created. Now the request will only be made at most once a second, and that often only if the user is a very slow typist.

Now let's look at the PHP side of things. Before looking at any code, suppose for a moment that you were working with a partner on this site, and he told you that he had already written the PHP validator; all you needed to do was access the page with method `GET` and two key/value pairs `password` and `password2`, each associated with the user's input in that field. Armed with that information, you would be able to write all of the above code without ever needing to look at the PHP. That's the beauty of abstraction, and it's how you'll be able to work with Google Maps on this problem set without seeing a single one of the thousands of lines of code that actually render the maps or let you pan and zoom.

```
<?php
    error_reporting(E_ALL ^ E_NOTICE ^ E_WARNING);
    header('Content-Type: text/xml');

    $password = $_GET["password"];
    $password2 = $_GET["password2"];
    if ($password != $password2)
    {
        print("<response valid=\"false\">Passwords must match</response>");
        exit;
    }
    if (strlen($password) < 6)
    {
        print("<response valid=\"false\">Password must be at least 6 characters long
               </response>");
        exit;
    }
    if (($fp = fopen("dictionary.txt", "r")) === FALSE)
        die("Could not open dictionary");
    $dictionary = array();
    while ((fscanf($fp, "%s", $word)) == 1)
```

[5]The code here is adapted from an example at http://www.selfcontained.us/2007/10/07/ajax-requests-when-users-stop-typing/.

[6]See how cool first-class functions are?

```
    {
        $dictionary[$word] = TRUE;
    }
    fclose($fp);
    if ($dictionary[strtolower($password)])
        print("<response valid=\"false\">Your password may not be a dictionary word
               </response>");
    else
        print("<response valid=\"true\">Your password is sufficiently strong
               </response>");
?>
```

You should be able to figure out this PHP on your own (hint: speller!). Pay special attention to the call to `header()`; this is necessary for the response to be interpreted as XML.

2.3 Validating a Username

There's no reason we should stop at passwords. After all, it's even more important that a user choose a valid username than a strong password—we might be able to accept a weak password or no password for a site requiring little security, but few database schemata will be happy with duplicate usernames. So let's look at a way to ensure that the user has chosen a unique username, again without any explicit submission on the user's part.

The JavaScript is omitted here because it barely changes at all—the interface it uses to validate usernames is nearly identical to the one it uses to validate passwords.

```
<?php
    require_once("includes/common.php");
    error_reporting(E_ALL ^ E_NOTICE ^ E_WARNING);
    header('Content-Type: text/xml');

    $username = $_GET["username"];

    if (strlen($username) < 6)
    {
        print("<response valid=\"false\">Username is not long enough</response>");
        exit;
    }

    $sql = "SELECT uid FROM users WHERE username = '"
           . mysql_real_escape_string($username) . "'";
    $result = mysql_query($sql);

    if (mysql_num_rows($result) == 1)
        print("<response valid=\"false\">That username is unavailable</response>");
    else
        print("<response valid=\"true\">That username is valid</response>");
?>
```

The changes in the PHP are more interesting. We're using code very similar to that used in login2.php to validate a user's login. Here we simply check if a query for a username returns any rows. If it does, the username is taken and the user will have to pick a different one.

While all this dynamic validation is fun to implement and use, remember that client-side validation is never a substitute for that done on the server side. There is no guarantee that a malicious (or simply

old-fashioned) user will not circumvent our JavaScript and submit invalid values for any field. Client-side validation serves mainly to create a more user-friendly experience and save the server a little bandwidth by enabling it to return small packets of data tailored to specific queries rather than having to process an entire form with potentially many incorrect inputs. It doesn't mean that the server can ever assume its inputs to be valid and sanitized.

3 Google Maps

Now we move on to the Google Maps API. While these notes will give an overview of the various tools Google provides to those who wish to make use of its maps, there's no substitute for going to the Google Maps API Developer Guide at http://code.google.com/apis/maps/documentation/index.html and reading about the API straight from the horse's mouth. Google also provides several individual examples that make clear the behavior of each aspect of the API. Try them out!

3.1 Including the API

First things first: before creating a page that uses the Google Maps API, be sure to include the following in your page's `<head>` element.

```
<script src="http://maps.google.com/maps?file=api&v=2&key=abcdefg&sensor=true_or_false"
        type="text/javascript">
</script>
```

You should replace `abcdefg` with the key you received when you signed up to use the GM API and `true_or_false` with `false`.[7]

3.2 Setting Up

Ordinarily you do most of your work within the body tag rather than with it, but in this case you'll need to modify it to ensure proper operation of the GM API.

```
<body onload="initialize()" onunload="GUnload()">
```

Can you guess what this does after the above tutorial on event handlers? When the body tag is loaded, i.e., when the page itself is loaded, the `initialize()` function will be called. Inside that function is where you're expected to create your map and prepare it for the user. You don't need to implement the `GUnload()` function yourself; that's a Google Maps API function designed to prevent memory leaks from stealing into your app and annoying your users by slowing down their browsers.

3.3 Creating a Map

The GMap2 object is at the core of the Maps API. The most basic way to create one is as follows:

```
var map = new GMap2(document.getElementById("map_canvas"));
```

The GMap2 constructor expects to be passed an element on the page, such as a `div`, where the map can be rendered. You may optionally pass a `GMap2Options` object as the second parameter if you want to customize your map further. In any case, you now have your map, albeit not a very useful one. This can be (partially) solved with a call to `setCenter()`.

```
map.setCenter(new GLatLng(42.376, -71.115), 16);
```

[7] ...unless you're doing something very interesting involving GPS for your final project, in which case you should choose `true`.

Look familiar? `setCenter()` takes a `GLatLng` object, which encapsulates a latitude and longitude, and a zoom level, which is simply a number representing how close or how far you want your view to be from the ground. If you'd rather have Google Maps tell you something about its latitude and longitude, a call like

```
var bounds = map.getBounds();
```

will place information about the four corners of the viewport in the `bounds` variable, enabling you to do something like

```
var southWest = bounds.getSouthWest();
```

to turn the `southWest` variable into a `GLatLng` with the latitude and longitude of the point at the lower left-hand corner of the viewport.

Feel free as well to use the `setMapType()` method to render the map type of your choice. The call

```
map.setMapType(MAPTYPE);
```

where `MAPTYPE` is one of `G_NORMAL_MAP` (for the standard street view), `G_SATELLITE_MAP` (for the photographic map), or `G_HYBRID_MAP` (for those who just can't decide) will change the rendering mode to the one of your choice.

3.4 Getting Dynamic

Now that you're satisfied with your map, you can make it do your bidding. The methods `setCenter()` and `panTo()` enable you to move the map to a different point programmatically, the former instantaneously and the latter smoothly. So something like

```
<script>
    function moveMap {
        map.panTo(new GLatLng(document.getElementById("lat").value,
        document.getElementById("lng").value), 13);
    }
</script>
<!--- body and map div here --->
<input id="lat" type="text"/><br/>
<input id="lng" type="text"/><br/>
<input type="button" value="Move" onclick="moveMap();"/>
```

would let your user fly smoothly around the world. We can also use the `openInfoWindowHtml()` method to pop up interesting facts about locations on the map:

```
map.openInfoWindowHtml(map.getCenter(),
                       document.createTextNode("This is the center of the map."));
```

Well, facts, anyway. Note the use of a DOM method to generate a text node out of the ether.

3.5 Events

The Google Maps API provides a somewhat more sophisticated way to bind events to handlers than the one you've seen so far. The `GEvent.addListener(object, event, function)` function binds the handler represented by `function` to the event specified in `event` for the given `object`. Different types of objects support different events.[8]

Event listeners bound in this way can also support closures. Remember the function used to throttle the number of AJAX queries from our username/password validator?

[8] You can see a complete list of events for all objects at http://code.google.com/apis/maps/documentation/reference.html.

```
function createMarker(point, number) {
    var marker = new GMarker(point);
    var message = ["This","is","the","secret","message"];
    marker.value = number;
    GEvent.addListener(marker, "click", function() {
        var myHtml = "<b>#" + number + "</b><br/>" + message[number -1];
        map.openInfoWindowHtml(point, myHtml);
    });
    return marker;
}

// Add 5 markers to the map at random locations
var bounds = map.getBounds();
var southWest = bounds.getSouthWest();
var northEast = bounds.getNorthEast();
var lngSpan = northEast.lng() - southWest.lng();
var latSpan = northEast.lat() - southWest.lat();
for (var i = 0; i < 5; i++) {
    var point = new GLatLng(southWest.lat() + latSpan * Math.random(),
    southWest.lng() + lngSpan * Math.random());
    map.addOverlay(createMarker(point, i + 1));
}
```

This code from the Google Maps API examples demonstrates how markers can be created with listener functions that reflect the values of outside data (the "messages" that they possess) even though the marker object itself does not know what the message is.

Like JavaScript events, Google Maps API events sometimes pass arguments giving information about the event. So

```
GEvent.addListener(map, "click", function(latlng) {
    if (latlng) {
        alert("You clicked the map at " + latlng.lat() + ", " + latlng.lng() + ".");
    }
}
```

would cause an alert with latitude/longitude data anytime someone clicked inside this map.

Finally, Google provides `GEvent.addDomListener()` for binding events like these for regular DOM objects in case you don't like doing it the usual way.

3.6 Controls

Controls are the objects on a map that allow you to interact with it. Controls include things like the zoom bar and the radio button-like map type selector. You can add a control with the `addControl()` function, which takes a control object to add:

```
map.addControl(new GLargeMapControl());
```

Dynamic controls like the map type selector, which has to know which map types are allowed before it can display itself, are constructed only once. As such, you need to iron out your map type before you add any controls. The `addControl()` function can take an additional argument, a `GControlPosition` object representing a corner of the map to create the control in and an offset from that corner:

```
map.addControl(new GLargeMapControl(),
               new GControlPosition(G_ANCHOR_TOP_RIGHT, new GSize(10,10));
```

3.7 Overlays

Map overlays are objects that are tied to points on the globe (as opposed to points in your viewport, like those to which controls are bound). The Maps API supports markers, lines of various types, and even the replacement of the map tile images themselves, but we'll be focusing on markers. Overlays in general are added with the `addOverlay()` function, which is passed an object to add as an overlay.

```
map.addOverlay(new GMarker(new GLatLng(100, 100)));
```

The constructor for a `GMarker` can accept a second parameter specifying options for the marker, like whether or not it should be draggable.

```
map.addOverlay(new GMarker(new GLatLng(50, 50), { draggable: true }));
```

Do you remember from lecture what type that last argument is? Another potential option is the type of icon the marker should use.

```
// Set up our marker to use a default icon
var blueIcon = new GIcon(G_DEFAULT_ICON);
blueIcon.image = "http://www.google.com/intl/en_us/mapfiles/ms/micons/blue-dot.png";
markerOptions = { icon:blueIcon };
map.addOverlay(new GMarker(point, markerOptions));

// Set up a marker to use a custom icon
var tinyIcon = new GIcon();
tinyIcon.image = "http://labs.google.com/ridefinder/images/mm_20_red.png";
tinyIcon.shadow = "http://labs.google.com/ridefinder/images/mm_20_shadow.png";
tinyIcon.iconSize = new GSize(12, 20);
tinyIcon.shadowSize = new GSize(22, 20);
tinyIcon.iconAnchor = new GPoint(6, 20);
tinyIcon.infoWindowAnchor = new GPoint(5, 1);
markerOptions = { icon:tinyIcon };
map.addOverlay(new GMarker(point2, markerOptions));
```

This Google sample code demonstrates setting up and adding a marker with a built-in graphic and a custom-defined graphic type, which is somewhat harder.

3.8 Services

The Google Maps API includes a few other features for utility purposes.

3.8.1 AJAX Requests and XML Parsing

If you don't want to deal with the standard `XMLHttpRequest` object, Google provides a cross-platform alternative in the `GXmlHttp` object. It's used exactly like an `XMLHttpRequest` other than the fact it must be created with a call to `GXmlHttp.create()` and not `new`. Google also provides an object that hides the process of waiting for `readyState == 4`, making the code for such a request much more compact.

```
GDownloadUrl("myfile.txt", function(data, responseCode) {
    alert(data);
});
```

If you want to avoid dealing with headers, the `GXml.parse()` method can convert a string into an XML object that can then be acted upon using normal DOM methods.

3.8.2 Geocoding

The incredibly useful `GClientGeocoder` object can convert string addresses like "1600 Pennsylvania Ave" into `GLatLng` objects, which are much more useful everywhere else in the API. The second argument to the `Geocoder`'s `getLatLng()` function is the function to be executed when the geocoding operation is complete, as it can be slow. The `GClientGeocoder` object also provides a `getLocations()` method that returns a JSON (JavaScript Object Notation) object with detailed information about an address. This can be used to convert the address typed by a user, which may be incorrect or missing information, into a canonical address with all information filled out. The `GClientGeocoder` can also perform *reverse geocoding*, returning a JSON object in response to a GLatLng object representing a point. Finally, much like Yahoo's stock quote service, the Google Maps API lets us request address data through a URL, http://maps.google.com/maps/geo?. The following arguments can be passed through the query string using a standard AJAX request:

- `q` (required)
  The address that you want to geocode.
- `key` (required)
  Your API key.
- `sensor` (required)
  Indicates whether or not the geocoding request comes from a device with a location sensor. This value must be either true or false.
- `output` (required)
  The format in which the output should be generated. The options are `xml`, `kml`, `csv`, or (default) `json`.
- `ll` (optional)
  The (latitude, longitude) of the viewport center expressed as a comma-separated string (e.g., `"ll=40.479581,-117.773438" )`. This parameter only has meaning if the `spn` parameter is also passed to the geocoder.
- `spn` (optional)
  The "span" of the viewport expressed as a comma-separated string of (latitude,longitude) (e.g., `"spn=11.1873,22.5"`). This parameter only has meaning if the `ll` parameter is also passed to the geocoder.
- `gl` (optional)
  The country code, specified as a ccTLD ("top-level domain") two-character value.