Lab 2: Programming with JavaScript

Learning Objectives:

- Tools for debugging JavaScript code
- Testing asynchronous operations
- Understanding JavaScript functions
- Closing in on closures

Debugging JavaScript code:

- There are two important approaches to debugging JavaScript: *logging* and *breakpoints*.

Logging

- Logging statements are part of the code.
- We can write logging calls in our code, and we can benefit from seeing the messages in the consoles of all modern browsers (Such as Firebug, Safari, Chrome, IE, and recent versions of Opera).
- For example, if we wanted to know what the value of a variable named x was at a certain point in the code, we might write this:

```
var x = 213;
console.log(x);
```

- An example of a simple logging method that works in all modern browsers can be:

```
function log() {
  try {
      console.log.apply(console, arguments);
  }
  catch(e) {
  try {
      opera.postError.apply(opera, arguments);
  }
  catch(e){
      alert(Array.prototype.join.call( arguments, " "));
      }
  }
}
```

- In the example above, we first try to log a message using the method that works in most modern browsers. If that fails, an exception will be thrown that we catch, and then we can try to log a message using Opera's proprietary method. If both of those methods fail, we fall back to using old-fashioned alerts.

Breakpoints
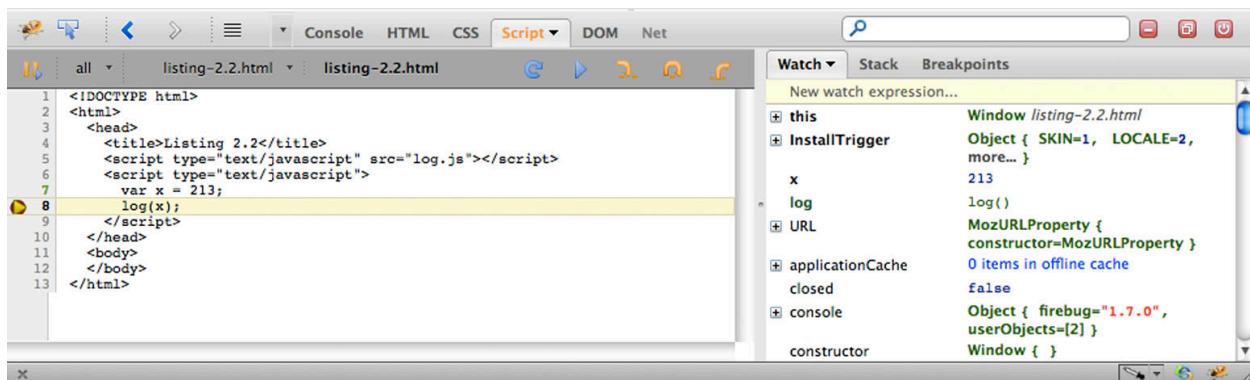
- Breakpoints are a somewhat more complex concept than logging
- Breakpoints halt the execution of a script at a specific line of code, pausing the browser.

- This allows us to investigate the state of all sorts of things at the point of the break. This includes all accessible variables, the context, and the scope chain.
- Let's say that we have a page that employs our new log() method as follows:

```html
<!DOCTYPE html>
<html>
  <head>
    <title>Listing 2.2</title>
    <script type="text/javascript" src="log.js"></script>
    <script type="text/javascript">
      var x = 213;
      log(x);                          Line where
    </script>                          we'll break
  </head>
  <body>
  </body>
</html>
```

- If we were to set a breakpoint using Firebug on the annotated line and refresh the page to cause the code to execute, the debugger would stop execution at that line and show us the following display:

- Note how the rightmost pane allows us to see the state within which our code is running, including the value of x.
- The debugger breaks on a line before the break-pointed line is actually executed; in this example, the call to the log() method has yet to be executed.
- If we were trying to debug a problem with our new method, we might want to step into that method to see what's going on inside it.
- Debugging code not only serves its primary and obvious purpose (detecting and fixing bugs), but it also can help us achieve the current best-practice goal of generating effective test cases.

Generating Tests

- Good tests exhibit three important characteristics:
  - Repeatability
  - Simplicity
  - Independence

- Two primary approaches for constructing tests are ***deconstructive*** and ***constructive*** tests.
    - Deconstructive test cases are created when existing code is deconstructed to isolate a problem, eliminating anything that's not relevant to the issue.
    - With a constructive test case, we start from a known good, reduced case and build up until we're able to reproduce the bug in question.
- For example, the following listing shows a simple DOM test case used to test jQuery.

```
<script src="dist/jquery.js"></script>
<script>
  $(document).ready(function() {
    $("#test").append("test");
});
</script>
<style>
  #test { width: 100px; height: 100px; background: red; }
</style>
<div id="test"></div>
```

- To generate a test, with a clean copy of the code base, we can use a little shell script to check out the library, copy over the test case, and build the test suite, as:

```
#!/bin/sh
# Check out a fresh copy of jQuery
git clone git://github.com/jquery/jquery.git $1
# Copy the dummy test case file in
cp $2.html $1/index.html
# Build a copy of the jQuery test suite
cd $1 && make
```

- Saved in a file named gen.sh, the preceding script would be executed using this command line:

```
$ ./gen.sh mytest dom
```

- This would pull in the DOM test case from dom.html in the Git repository.
- Another alternative is to use a prebuilt service designed for creating simple test cases
    - One of these services is JS Bin (http://jsbin.com/)
- Popular testing frameworks include:
    - QUnit
    - YUI Test
    - JsUnit
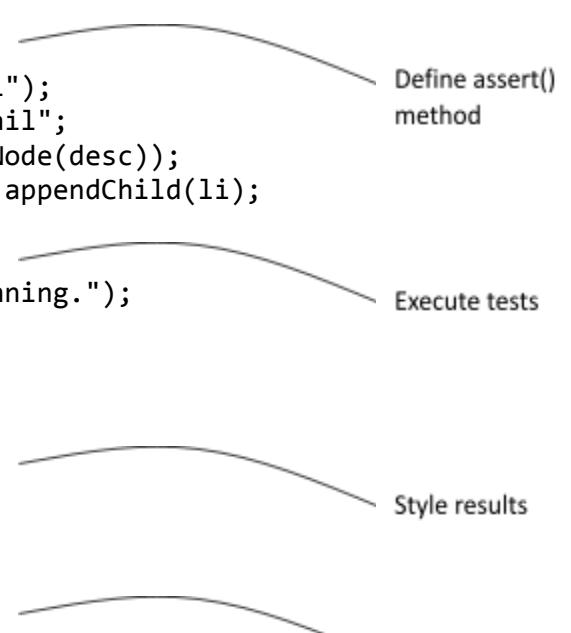
About Test Suits

- The primary purpose of a test suite is to aggregate all the individual tests that your codebase might have into a single unit so that they can be run in bulk.
- The core of a unit-testing framework is its assertion method, usually named assert().
- This method usually takes a value—an expression whose premise is asserted—and a description that describes the purpose of the assertion. If the value evaluates to true, and in other words is "truthy," the assertion passes; otherwise, it's considered a failure.
- Example of a simple implementation of a JavaScript assertion:

```html
<html>
<head>
  <title>Test Suite</title>
  <script>
    function assert(value, desc) {              Define assert()
    var li = document.createElement("li");      method
    li.className = value ? "pass" : "fail";
    li.appendChild(document.createTextNode(desc));
    document.getElementById("results").appendChild(li);
    }
    window.onload = function() {
    assert(true, "The test suite is running.");  Execute tests
    assert(false, "Fail!");
    };
  </script>
  <style>
    #results li.pass { color: green; }
    #results li.fail { color: red; }           Style results
  </style>
</head>
<body>
  <ul id="results"></ul>                        Hold test results
</body>
</html>
```

- Simple assertions are useful, but they really begin to shine when they're grouped together in a testing context to form **test groups**.
- A test group is built in a way such that individual assertions are inserted into the results. If any assertion fails, then the entire test group is marked as failing.

Asynchronous testing

- Asynchronous tests are tests whose results will come back after a nondeterministic amount of time has passed; common examples of this situation are Ajax requests and animations.
- To handle asynchronous tests, we need to follow a couple of simple steps:
  - o Assertions that rely upon the same asynchronous operation need to be grouped into a unifying test group.
  - o Each test group needs to be placed on a queue to be run after all the previous test groups have finished running.
  - o Thus, each test group must be capable of running asynchronously.
  - o An example of a simple asynchronous test suit is as follows:

```html
<html>
  <head>
    <title>Test Suite</title>
    <script>
      (function() {
```

```
      var queue = [], paused = false, results;
      this.test = function(name, fn) {
        queue.push(function() {
        results = document.getElementById("results");
        results = assert(true, name).appendChild(
          document.createElement("ul"));
        fn();
      });
      runTest();
    };
    this.pause = function() {
      paused = true;
    };
    this.resume = function() {
      paused = false;
      setTimeout(runTest, 1);
    };
    function runTest() {
      if (!paused && queue.length) {
        queue.shift()();
        if (!paused) {
          resume();
        }
      }
    }
    this.assert = function assert(value, desc) {
      var li = document.createElement("li");
      li.className = value ? "pass" : "fail";
      li.appendChild(document.createTextNode(desc));
      results.appendChild(li);
      if (!value) {
        li.parentNode.parentNode.className = "fail";
      }
      return li;
    };
  })();
  window.onload = function() {
    test("Async Test #1", function() {
      pause();
      setTimeout(function() {
        assert(true, "First test completed");
        resume();
      }, 1000);
    });
    test("Async Test #2", function() {
      << #### Write Code here #### >>
    });
  };
</script>
```

- The one internal implementation function, runTest(), is called whenever a test is queued or dequeued. It checks to see if the suite is currently unpaused and if there's something in the queue, in which case it'll dequeue a test and try to execute it. Additionally, after the test group is finished executing, runTest() will check to see if the suite is currently paused, and if it's not (meaning that only asynchronous tests were run in the test group), runTest() will begin executing the next group of tests.

Exercise 1: Complete the Asynchronous Test Suit

- Complete the in incomplete code in the listing above

Understanding JavaScript functions:

- In JavaScript, functions can be created via literals.
- Functions can be assigned to variables, array entries, and properties of other objects.
- Functions can be passed as arguments to functions.
- Functions can be returned as values from functions.
- Functions can possess properties that can be dynamically created and assigned.

The browser event loop

- The following types of events can occur, among others:
    o Browser events, such as when a page is finished loading
    o Network events, such as responses to an Ajax request
    o User events, such as mouse clicks, mouse moves, or keypresses
    o Timer events, such as when a timeout expires or an interval fires
- Browser event loop is single-threaded
- Every event that's placed into the event queue is handled in FIFO order.
- Event handlers are examples of a more general concept known as callback functions.

Callback Functions

- A function that some other code will later "call back" into at an appropriate point of execution
- An example callback function that shows the ability to pass a function as an argument to another function

```
function test_function(callback) { return callback(); }
var text = 'Domo arigato!';
assert(test_function(function(){ return text; }) === text,
      "The useless function works! " + text);
```

Declarations

- Functions are declared with:
    o The function keyword.

6

  o An optional name that, if specified, must be a valid JavaScript identifier.

  o A comma-separated list of parameter names enclosed in parentheses.

  o The body of the function, as a series of JavaScript statements enclosed in braces.

Exercise 2: Explain the above listing

- Verbally explain what the above listing does. Which are named, and which are anonymous functions here?

Scoping and functions
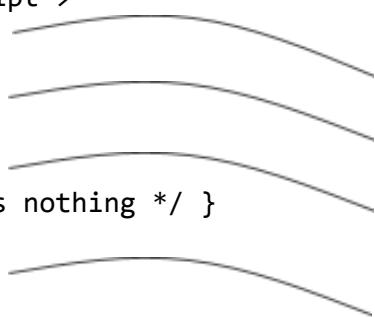
- Variable declarations are in scope from their point of declaration to the end of the function within which they're declared, regardless of block nesting.
- Named functions are in scope within the entire function within which they're declared, regardless of block nesting.
- For the purposes of declaration scopes, the global context acts like one big function encompassing the code on the page.
- Example of function scopes:

Considering the following to be a Test block:

```
assert(true,"some descriptive text");
assert(typeof outer==='function',
"outer() is in scope");
assert(typeof inner==='function',
"inner() is in scope");
assert(typeof a==='number',
"a is in scope");
assert(typeof b==='number',
"b is in scope");
assert(typeof c==='number',
"c is in scope");
```

Observing the scoping behavior of declarations:

```
<script type="text/javascript">
  /* test code here */
  function outer(){
  /* test code here */
  var a = 1;
  /* test code here */
  function inner(){ /* does nothing */ }
  var b = 2;
  /* test code here */
  if (a == 1) {
    var c = 3;
    /* test code here */
```

Running test block before defining anything

Running test block after defining function outer()

Running test block inside function outer()

Running test block after inner()

```
  }
  /* test code here */
}
outer();
/* test code here */
</script>
```

Running test block inside function outer() but after the if block is closed.

Function Parameters and Invocations

- If there is a different number of arguments than there are parameters, no error is raised.
- The *arguments* parameter is a collection of all of the arguments passed to the function.
- The *this* parameter refers to an object that's implicitly associated with the function invocation and is termed the function context.
- **Invocation as a method**

```
var o = {};
o.whatever = function(){};
o.whatever();
```

- **Invocation as a constructor**

```
function constr(){ return this; }
new constr();
```

- **Invocation with the apply() and call() methods**

```
<script type="text/javascript">
  function juggle() {
    var result = 0;
    for (var n = 0; n < arguments.length; n++) {
      result += arguments[n];
    }
   this.result = result;
  }
  var varTest1 = {};
  var varTest2 = {};
  juggle.apply(varTest1,[1,2,3,4]);
  juggle.call(varTest2,5,6,7,8);
  assert(varTest1.result === 10,"juggled via apply");
  assert(varTest2.result === 26,"juggled via call");
</script>
```

- **Forcing the function context in callbacks**

```
<script type="text/javascript">
  function forEach(list,callback) {
    for (var n = 0; n < list.length; n++) {
      callback.call(list[n],n);
    }
```

```
    }
    var weapons = ['shuriken','katana','nunchucks'];
    forEach(weapons,function(index){
      assert(this == weapons [index],
        "Got the expected value of " + weapons [index]);
      }
    );
</script>
```
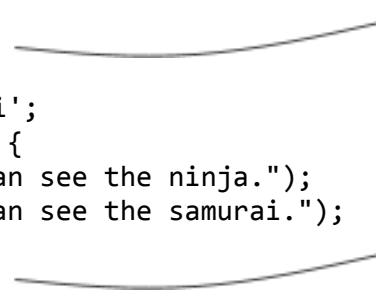
Closing in on closures

- A closure is the scope created when a function is declared that allows the function to access and manipulate variables that are external to that function.
- An example of closure is as follows:

```
<script type="text/javascript">
  var outerValue = 'ninja';
  var later;
  function outerFunction() {
    var innerValue = 'samurai';
    function innerFunction() {
      assert(outerValue,"I can see the ninja.");
      assert(innerValue,"I can see the samurai.");
    }
    later = innerFunction;
  }
  outerFunction();
  later();
</script>
```

Variable later declared is in global scope

Stores a reference to the innerFunction() to the later variable.

- Closures create a "safety bubble" that contains the function and its variables and stays around as long as the function itself does.
- Function parameters are included in the closure of that function.
- All variables in an outer scope, even those declared after the function declaration, are included.
- However, within the same scope, variables not yet defined cannot be forward-referenced.
- A common use of closures is to encapsulate some information as a "private variable" (i.e., to limit the scope of such variables).

Callbacks and timers

- In both cases, a function is being asynchronously called at an unspecified later time, and within such functions, we frequently need to access outside data.
- Example: Using closures from a callback for an Ajax request:

```
<div id="testSubject"></div>
<button type="button" id="testButton">Go!</button>
<script type="text/javascript">
```

```
  jQuery('#testButton').click(function(){
    var elem$ = jQuery("#testSubject");
    elem$.html("Loading...");
    jQuery.ajax({
      url: "test.html",
      success: function(html){
            assert(elem$, "We can see elem$, via the closure for this
callback.");
        elem$.html(html);
      }
    });
  });
</script>
```

- Using the $ sign as a suffix or prefix is a jQuery convention to indicate that the variable holds a jQuery object reference.

- Example: Using a closure in a timer interval callback

```
<div id="test-box"> Test-Element </div>
<script type="text/javascript">
  function animateIt(elementId) {
    var elem = document.getElementById(elementId);
    var tick = 0;
    var timer = setInterval(function(){
      if (tick < 100) {
        elem.style.left = elem.style.top = tick + "px";
        tick++;
      }
      else {
      ...
      << Write Code here >>
      ...
      }
    }, 10);
}
animateIt('test-box');
</script>
```

Exercise 3: Complete the Above Listing

- Complete the code inside the else condition in the above example. The code should do the following:

  o After 100 ticks, we stop the timer and perform tests to assert that we can see all relevant variables to perform the animation.

  o Hint: use the clearInterval() and assert() methods.

Overriding function behavior

- **Memoization:** It is the process of building a function that is capable of remembering its previously computed answers.

- Example of Memoizing functions using closures

```
<script type="text/javascript">
  Function.prototype.memoized = function(key){
      this._values = this._values || {};
      return this._values[key] !== undefined ?
        this._values[key] :
        this._values[key] = this.apply(this, arguments);
  };
  Function.prototype.memoize = function(){
      var fn = this;
      return function(){
        return fn.memoized.apply( fn, arguments );
      };
  };
  var isPrime = (function(num) {
      var prime = num != 1;
      for (var I = 2; I < num; i++) {
        if (num % I == 0) {
          prime = false;
          break;
        }
      }
  return prime;
}).memoize();
assert(isPrime(17),"17 is prime");
</script>
```

- **Function wrapping:** Function wrapping is a technique for encapsulating the logic of a function while overwriting it with new or extended functionality in a single step.

- Example of wrapping an old function with a new piece of functionality

```
function wrap(object, method, wrapper) {
  var fn = object[method];
      return object[method] = function() {
        return wrapper.apply(this, [fn.bind(this)].concat(
          Array.prototype.slice.call(arguments)));
      };
  }
if (Prototype.Browser.Opera) {
  wrap(Element.Methods, "readAttribute",
      function(original, elem, attr) {
        return attr == "title" ? elem.title : original(elem, attr);
      });
```

```
}
```

Immediate functions

- Immediate functions are anonymous functions that can have a closure just like any other function.
- They also have access to all the outside variables and parameters that are in the same scope as the statement during the brief life of the function.
- Immediate functions can be written as:

```
(function(){
  statement-1;
  statement-2;
  ...
  statement-n;
})();
```

- They can be used to create temporary scopes, as such a function is executed immediately, and, as with all functions, all the variables inside of it are confined to its inner scope. For example:

```
(function(){
  var numClicks = 0;
  document.addEventListener("click", function(){
  alert( ++numClicks );
  }, false);
})();
```

- Some common uses of immediate functions are:
    - Loops
    - Library Wrapping

Exercise 4: Explain the Code Below

- Verbally explain what is happening in the code below. Which one is an immediate function? How is the immediate function helping in the code?

```
<div>DIV 0</div>
<div>DIV 1</div>
<script type="text/javascript">
  var div = document.getElementsByTagName("div");
  for (var i = 0; i < div.length; i++) (function(n){
      div[n].addEventListener("click", function(){
        alert("div #" + n + " was clicked.");
      }, false);
  })(i);
</script>
```