# HTML and JavaScript Typical Setup and Usage

|  |
| --- |
| ***javascriptdemo.html***  <!DOCTYPE html>  <html>  <head>  <title>JavaScript Demo</title>  <meta charset="utf-8" />  </head>  <body>  <header><h1>JavaScript Demo</h1></header>  <script src="myjs.js"></script>  </body>  </html> |
| ***myjs.js***  var h1s = document.getElementsByTagName('h1');  h1s[0].style.backgroundColor = "red"; |
|  |

# Functions

1. ***Write a function*** that accepts an array of numbers as its only parameter. The function returns the largest number in the array.
   1. Make up a test case and then show that the function works

|  |
| --- |
| ***javascriptdemo.html***  <!DOCTYPE html>  <html>  <head>  <title>JavaScript Demo</title>  <meta charset="utf-8" />  </head>  <body>  <header><h1>JavaScript Demo</h1></header>  <script src="myjs.js"></script>  **<script src="q1.js"></script>**  </body>  </html> |
| ***q1.js***  **function getLargestNumber(numbers) {**  **var largest = numbers[0];**  **for (var i = 1; i < numbers.length; ++i) {**  **if (numbers[i] > largest) {**  **largest = numbers[i];**  **}**  **}**  **return largest;**  **}**  **var largest = getLargestNumber([56, 23, 76, 12, 8]);**  **console.log(largest); // should be 76** |
|  |

1. ***Write a function as an expression*** that accepts an array of numbers as its only parameter. The function returns the average of the numbers.
   1. Make up a test case and then show that the function works

|  |
| --- |
| ***q2.js***  **var getAverage = function (numbers) {**  **var total = 0;**  **for(var value of numbers) {**  **total += value;**  **}**  **return total / numbers.length;**  **};**  **var average = getAverage([56, 23, 76, 12, 8]);**  **console.log("Average = " + average); // should be 35** |
|  |

1. Write a function that accepts an object as its only parameter. The object has the following properties: name, age, height, weight, telephone, email. The function simply logs all the properties to the console.
   1. Demonstrate that the function works.

|  |
| --- |
| ***q3.js***  **function logPerson(person) {**  **for (var key in person) {**  **console.log(key + " --> " + person[key]);**  **}**  **};**  **logPerson({**  **name: "Jeff",**  **age: 54,**  **height: 1.9,**  **weight: 200,**  **telephone: "423-222-1111",**  **email: "abc@abc.com"**  **});** |
|  |

1. Write a function called ‘areaOfCircle’ that calculates and returns the area of a circle (). The function takes no parameters but should assume that there is an object with property ‘radius’ (I.e. use this.radius).
   1. Create object called circle1 with radius 10.
   2. Create an object called circle2 with radius 100.
   3. Use context binding to output the area of circle1 and circle2.

|  |
| --- |
| ***q4.js***  **function areaOfCircle() {**  **return Math.PI \* this.radius \* this.radius;**  **}**  **var circle1 = { radius: 10 };**  **var circle2 = { radius: 100 };**  **console.log(areaOfCircle.call(circle1));**  **console.log(areaOfCircle.call(circle2));** |
|  |

1. Use closures to implement a function that returns a unique string each time it is called.
   1. Demonstrate the function

|  |
| --- |
| ***q5.js***  **var uniqueString = (function() {**  **var value = 0;**  **return function () {**  **var string = "String" + value;**  **value = value + 1;**  **return string;**  **};**  **})();**  **for (var count = 0; count < 10; count++) {**  **console.log(uniqueString());**  **}** |
|  |

1. Use closures to generate 10 links. Each link’s text should read ‘Link <n>’, where <n> is a multiple of 10, starting at 10 and ending at 100. When the link is clicked, change an area on the page to read ‘You clicked link <n>” where n is the link’s number.
   1. Demonstrate that it works

|  |
| --- |
| ***javascriptdemo.html***  <!DOCTYPE html>  <html>  <head>  <title>JavaScript Demo</title>  <meta charset="utf-8" />  </head>  <body>  <header><h1>JavaScript Demo</h1></header>  **<p id="linkArea"></p>**  **<p id="outputArea"></p>**  <script src="myjs.js"></script>  <script src="q1.js"></script>  <script src="q2.js"></script>  <script src="q3.js"></script>  <script src="q4.js"></script>  <script src="q5.js"></script>  **<script src="q6.js"></script>**  </body>  </html> |
| ***q6.js***  **var createLink = function (text) {**  **var a = document.createElement("a");**  **var textNode = document.createTextNode(text);**  **a.appendChild(textNode);**  **a.href = "#";**  **return a;**  **};**  **var getLinkText = (function () {**  **var value = 10;**  **return function () {**  **var string = "Link " + value;**  **value = value + 10;**  **return string;**  **};**  **})();**  **var linkArea = document.getElementById("linkArea");**  **for (var count = 0; count < 10; count++) {**  **var link = createLink(getLinkText());**  **link.onclick = function () {**  **var outputArea = document.getElementById("outputArea");**  **outputArea.innerHTML = "You clicked " + this.innerHTML;**  **};**  **linkArea.appendChild(link);**  **linkArea.appendChild(document.createElement("br"));**  **}** |
|  |

# Objects

1. Create an object called ‘**car**’ with the following properties:
   1. **yearModel**: a string that holds the car’s model year
   2. **make**: a string that holds the car’s make
   3. **speed**: a number that holds the car’s current speed
   4. **accelerate**: a method that adds 5 to the **speed** property
   5. **brake**: a method that subtracts 5 from the **speed** property
   6. Demonstrate the object in action
   7. Demonstrate how to remove the **make** property from **car**

|  |
| --- |
| ***q7.js***  **var car = {**  **yearModel: "1970",**  **make: "Honda",**  **speed: 0,**  **accelerate: function () {**  **this.speed += 5;**  **},**  **brake: function () {**  **this.speed -= 5;**  **}**  **};**  **console.log(car);**  **car.accelerate();**  **car.accelerate();**  **console.log(car);**  **car.brake();**  **console.log(car);**  **delete car.make;**  **console.log(car);** |
|  |

1. Create a constructor function called ‘**Car**’. The function should take 3 parameters: **yearModel**, **make**, and **speed**.
   1. **yearModel**: a private string that holds the car’s model year
   2. **make**: a private string that holds the car’s make
   3. **speed**: a private number that holds the car’s current speed
   4. Use the constructor function to create 3 different instances of Car and then log each instance**.**
   5. Call accelerate on one of the instances.
   6. Use prototyping to add the **accelerate** and **brake** methods.
      1. **accelerate**: a method that adds 5 to the **speed** property
      2. **brake**: a method that subtracts 5 from the **speed** property
   7. Call the **accelerate** and **brake** methods on the instances and then output the **speed** value of each instance.

|  |
| --- |
| ***q8.js***  **var Car = function (yearModel, make, speed) {**  **var \_yearModel = yearModel;**  **var \_make = make;**  **var \_speed = speed;**  **this.getYearModel = function () {**  **return \_yearModel;**  **};**  **this.getMake = function () {**  **return \_make;**  **};**  **this.getSpeed = function () {**  **return \_speed;**  **};**  **this.setSpeed = function (speed) {**  **\_speed = speed;**  **};**  **};**  **var car1 = new Car("1990", "Honda", 0);**  **var car2 = new Car("2001", "Dodge", 10);**  **var car3 = new Car("2010", "Toyota", 15);**  **console.log(car1);**  **console.log(car2);**  **console.log(car3);**  **//car1.accelerate();**  **Car.prototype.accelerate = function () {**  **this.setSpeed(this.getSpeed() + 5);**  **};**  **Car.prototype.brake = function () {**  **this.setSpeed(this.getSpeed() - 5);**  **};**  **car1.accelerate();**  **console.log(car1.getSpeed());**  **car2.brake();**  **car2.brake();**  **console.log(car2.getSpeed());** |
|  |

1. Use constructor functions, prototyping, closures, and context binding to implement the following class diagram. Use a separate file to define the classes.

|  |
| --- |
|  |

* 1. Instantiate an **Employee** and output its values.
     1. Try outputting the employee’s pay (call **getPay**)
  2. Instantiate a **ProductionWorker** and output its values. Output the worker’s pay after working for 10 hours.

|  |
| --- |
| ***javascriptdemo.html***  <!DOCTYPE html>  <html>  <head>  <title>JavaScript Demo</title>  <meta charset="utf-8" />  </head>  <body>  <header><h1>JavaScript Demo</h1></header>  <p id="linkArea"></p>  <p id="outputArea"></p>  <script src="myjs.js"></script>  <script src="q1.js"></script>  <script src="q2.js"></script>  <script src="q3.js"></script>  <script src="q4.js"></script>  <script src="q5.js"></script>  <script src="q6.js"></script>  <script src="q7.js"></script>  <script src="q8.js"></script>  **<script src="Employee.js"></script>**  **<script src="q9.js"></script>**  </body>  </html>  </html> |
| ***Employee.js***  **var Employee = function (name, id, hireDate) {**  **var \_name = name;**  **var \_id = id;**  **var \_hireDate = hireDate;**  **this.getName = function () {**  **return \_name;**  **};**  **this.getId = function () {**  **return \_id;**  **};**  **this.getHireDate = function () {**  **return \_hireDate;**  **};**  **};**  **Employee.prototype.getPay = function () {**  **return 0;**  **};**  **var ProductionWorker = function (name, id, hireDate, shift, hourlyPayRate) {**  **Employee.call(this, name, id, hireDate);**  **var \_shift = shift;**  **var \_hourlyPayRate = hourlyPayRate;**  **this.getShift = function () {**  **return \_shift;**  **};**  **this.getHourlyPayRate = function () {**  **return \_hourlyPayRate;**  **};**  **};**  **ProductionWorker.prototype = new Employee();**  **ProductionWorker.prototype.getPay = function (hours) {**  **return hours \* this.getHourlyPayRate();**  **};** |
| ***q9.js***  **(function () {**  **var emp = new Employee("Jeff", "12", new Date());**  **console.log(emp.getName());**  **console.log(emp.getId());**  **console.log(emp.getHireDate());**  **console.log(emp.getPay());**  **var worker = new ProductionWorker("Jeffrey", "32", new Date("January 13, 2014"), 1, 10.87);**  **console.log(worker.getName());**  **console.log(worker.getId());**  **console.log(worker.getHireDate());**  **console.log(worker.getPay(10));**  **})();** |
|  |

QUnit

# What is QUnit?

* Client-side unit-testing framework
* Similar to server-side frameworks (NUnit, JUnit)
* Built by the jQuery team to test jQuery features
* No dependencies
* Can be used to test server-side JavaScript

# Setting up QUnit

* <http://qunitjs.com/>
* <https://github.com/jquery/qunit>
* Download the js and css files
* Use the CDN

# QUnit Test Runner

|  |
| --- |
| **<!DOCTYPE html>**  **<html>**  **<head>**  **<title>Sample Test Runner</title>**  **<meta charset="utf-8" />**  **<link rel="stylesheet" href="https://code.jquery.com/qunit/qunit-1.23.0.css" />**  **<script src="https://code.jquery.com/qunit/qunit-1.23.0.js"></script>**  **<script src="sut.js"></script>**  **<script src="tests.js"></script>**  **</head>**  **<body>**  **<div id="qunit"></div>**  **<div id="qunit-fixture">Test Markup</div>**  **</body>**  **</html>** |
|  |

# Simple First Test

|  |
| --- |
| ***tests.js***  **test('my first test', function () {**  **ok(true);**  **});** |
|  |

# Assertions

* ok(state[, message])
  + A boolean check. Passes if the first argument is truthy.
* equal(actual, expected[, message])
* notEqual(actual, expected[, message])
  + Non-strict comparisons.
* deepEqual(actual, expected[, message])
  + A deep recursive comparison, working on primitive types, arrays, objects, regular expressions, dates and functions.
* notDeepEqual(actual, expected[, message])
  + An inverted deep recursive comparison, working on primitive types, arrays, objects, regular expressions, dates and functions.
* strictEqual(actual, expected[, message])
  + A strict type and value comparison.
* notStrictEqual(actual, expected[, message])
  + A strict comparison, checking for inequality.
* propEqual(actual, expected[, message])
  + A strict type and value comparison of an object’s own properties.
* notPropEqual(actual, expected[, message])
  + A strict comparison of an object’s own properties, checking for inequality.
* throws(block[, expected][, message])
  + Test if a callback throws an exception, and optionally compare the thrown error

# Modules

|  |
| --- |
| module('module 1');  test('my first test', function(){  ok(true);  });  module('module 2');  test('my second test', function(){  ok(false);  }); |
| module('module 1', {  setup: function() {  },  teardown: function(){  }  });  test('my first test', function(){  ok(true);  }); |

# Synchronous Callbacks

Occasionally, callback assertions in the code might not be called at all, causing the test to fail silently.

QUnit provides a special assertion to define the number of assertions a test contains. If the test completes without the expected number of assertions, it will fail.

In order to indicate the expected number of assertion, call assert.expect() at the start of a test, with the number of expected assertions as the only argument.

|  |
| --- |
| QUnit.test("a test with synchronous callback", function (assert) {  assert.expect(2);  function calc(x, operation) {  return operation(x);  }  var result = calc(2, function (x) {  assert.ok(true, "calc() calls operation function");  return x \* x;  });  assert.equal(result, 4, "2 square equals 4");  }); |
| ***Practical Example (Uses JQuery)***  QUnit.test( "a test", function( assert ) {  assert.expect( 1 );    var $body = $( "body" );    $body.on( "click", function() {  assert.ok( true, "body was clicked!" );  });    $body.trigger( "click" );  }); |

assert.expect() provides the most value when actually testing callbacks. When all code is running in the scope of the test function, assert.expect() provides no additional value—any error preventing assertions to run would cause the test to fail anyway, because the test runner catches the error and fails the unit.

# Asynchronous Callbacks

While assert.expect() is useful to test synchronous, it can fall short for asynchronous callbacks. Asynchronous callbacks conflict with the way the test runner queues and executes tests. When code under test starts a timeout or interval or an AJAX request, the test runner will just continue running the rest of the test, as well as other tests following it, instead of waiting for the result of the asynchronous operation.

For every asynchronous operation in your QUnit.test() callback, use assert.async(), which returns a "done" function that should be called when the operation has completed.

|  |
| --- |
| QUnit.test( "two async calls", function( assert ) {  assert.expect( 2 );    var done1 = assert.async();  var done2 = assert.async();  setTimeout(function() {  assert.ok( true, "test resumed from async operation 1" );  done1();  }, 500 );  setTimeout(function() {  assert.ok( true, "test resumed from async operation 2" );  done2();  }, 150);  }); |
| QUnit.test("multiple call done()", function (assert) {  assert.expect(3);  var done = assert.async(3);  setTimeout(function () {  assert.ok(true, "first call done.");  done();  }, 500);  setTimeout(function () {  assert.ok(true, "second call done.");  done();  }, 500);  setTimeout(function () {  assert.ok(true, "third call done.");  done();  }, 500);  }); |