CS 146: Intro to Web Programming and Project Development

Instructor: Iraklis Tsekourakis

Lieb 213

Email: itsekour@stevens.edu

JavaScript

The DOM

- **D**ocument **O**bject **M**odel: a tree of the entire contents of your site
- Typically text nodes are leaves while tags containing data are internal nodes
- Through JS it is possible to navigate and modify everything in the DOM, from existing tags to attributes of such tags
- To access the DOM we use the document object
- For example you can change the background color of a page doing document.bgColor=“Salmon”;

The Images Array

- The document object contains an array called images with every image on your site (in order)
- You can access these images by doing
  - document.images[index]
  - document.images[name] // assuming you used a name attribute for that image
- Once you are referring to an image, you can change every attribute of it by simply doing .attr for example
  - document.images[name].src = “newimage.png”;

The Links Array

- As with the images array, all links on your page can be referred through the links array either by index or by name
- You can also change any attribute of a link you wish
  - document.links[n].href = “http://www.stevens.edu”;

The Forms Array

- All forms (in case you had more than one) can be accessed like the images and links array
- Once you access a form, you can refer to any input in it by adding .name, you can also access those items' attributes by adding .attr
  - document.forms[0].name.value = "Bob";
- This makes it very easy to extract the contents of a form
- To check a radio button (since they have the same, interpret the name as an array)
  - document.forms[0].rad[0].checked = true; // this will force the first radio button to be selected

Accessing Select Areas

- The options of a select input are placed in an array called options, and the currently selected item will be in the attribute selectedIndex
- To access the value of an option use .value, for the text use .text
- e.g.
  - var sel = document.forms[0].sel; // we store in a var so we don't have to type it all again
  - var selIndex = sel.selectedIndex;
  - console.log("Currently selected item has value: " + sel.options[selIndex].value);

JS Error Handling

- When executing JavaScript code, different errors can occur
- Errors can be coding errors made by the programmer, errors due to wrong input, and other unforeseeable things
- The **try** statement lets you test a block of code for errors
- The **catch** statement lets you handle the error
- The **throw** statement lets you create custom errors
- The **finally** statement lets you execute code, after try and catch, regardless of the result

Try-Catch

- The **try** statement allows you to define a block of code to be tested for errors while it is being executed
- The **catch** statement allows you to define a block of code to be executed, if an error occurs in the try block
- The JavaScript statements **try** and **catch** come in pairs:

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
```

Try-Catch Example

```
<p id="demo"></p>

<script>
try {
    adddlert("Welcome guest!");
}
catch(err) {
    document.getElementById("demo").innerHTML = err.message;
}
</script>
```

The Throw Statement

- When an error occurs, JavaScript will normally stop and generate an error message
- The technical term for this is: JavaScript will **throw an exception (throw an error)**
- JavaScript will actually create an **Error object** with two properties: **name** and **message**
- The **throw** statement allows you to create a custom error
- Technically you can **throw an exception (throw an error)**

The Throw Statement Example

- The exception can be a JavaScript String, a Number, a Boolean or an Object:

```
x = document.getElementById("demo").value;
try {
    if(x == "") throw "empty";
    if(isNaN(x)) throw "not a number";
    x = Number(x);
    if(x < 5) throw "too low";
    if(x > 10) throw "too high";
}
catch(err) {
    message.innerHTML = "Input is " + err;
}
```

The finally Statement

- The **finally** statement lets you execute code, after try and catch, regardless of the result:

```
try {
    Block of code to try
}
catch(err) {
    Block of code to handle errors
}
finally {
    Block of code to be executed regardless of the try / catch result
}
```

The Finally Statement Example

```
<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>

<p id="message"></p>

<script>
function myFunction() {
    var message, x;
    message = document.getElementById("message");
    message.innerHTML = "";
    x = document.getElementById("demo").value;
    try {
        if(x == "")  throw "is empty";
        if(isNaN(x)) throw "is not a number";
        x = Number(x);
        if(x > 10)   throw "is too high";
        if(x < 5)    throw "is too low";
    }
    catch(err) {
        message.innerHTML = "Input " + err;
    }
    finally {
        document.getElementById("demo").value = "";
    }
}
</script>
```

JS Debugging

- It is difficult to write JavaScript code without a debugger.
- Your code might contain syntax errors, or logical errors, that are difficult to diagnose
- Often, when JavaScript code contains errors, nothing will happen; there are no error messages, and you will get no indications where to search for errors
- Normally, errors will happen, every time you try to write some new JavaScript code.
- Debugging is not easy, but fortunately, all modern browsers have a built-in debugger (console)

Debugging and Breakpoints

- The **debugger** keyword stops the execution of JavaScript, and calls (if available) the debugging function
- This has the same function as setting a breakpoint in the debugger
- If no debugging is available, the debugger statement has no effect
- With the debugger turned on, this code will stop executing before it executes the third line

```
var x = 15 * 5;
debugger;
document.getElementbyId("demo").innerHTML = x;
```

JS Strict Mode

- Execution using strict mode
- The purpose of "use strict" is to indicate that the code should be executed in "strict mode"
- With strict mode, you can not, for example, use undeclared variables
- Strict mode is declared by adding "use strict"; to the beginning of a script or a function
- Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

```
"use strict";
x = 3.14;       // This will cause an error because x is not declared
```

Why Strict Mode?

- Strict mode makes it easier to write "secure" JavaScript
- Strict mode changes previously accepted "bad syntax" into real errors
- As an example, in normal JavaScript, mistyping a variable name creates a new global variable; in strict mode, this will throw an error, making it impossible to accidentally create a global variable
- In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties
- In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error

JS: Avoid Global Variables

- Minimize the use of global variables
- This includes all data types, objects, and functions
- Global variables and functions can be overwritten by other scripts
- Use local variables instead, and learn how to use closures
- All variables used in a function should be declared as **local** variables
- Local variables **must** be declared with the **var** keyword, otherwise they will become global variables (unless strict)

JS: Declarations on Top & Initialization

- It is a good coding practice to put all declarations at the top of each script or function
  - Give cleaner code
  - Provide a single place to look for local variables
  - Make it easier to avoid unwanted (implied) global variables
  - Reduce the possibility of unwanted re-declarations
- It is a good coding practice to initialize variables when you declare them
  - Give cleaner code
  - Provide a single place to initialize variables
  - Avoid undefined values

JS: Best Practices

- Always treat numbers, strings, or booleans as primitive values, not as objects
- Declaring these types as objects, slows down execution speed, and produces nasty side effects:

```
var x = "John";
var y = new String("John");
(x === y) // is false because x is a string and y is an object.
```

- Beware that numbers can accidentally be converted to strings or NaN (Not a Number)
  - JavaScript is loosely typed; a variable can contain different data types, and a variable can change its data type
- Use === Comparison
  - The == comparison operator always converts (to matching types) before comparison; the === operator forces comparison of values and type

JS: Common Mistakes

- Confusing =, ==, ===
- Confusing Addition and Concatenation (+)
- Breaking a Return statement
- Undefined is NOT Null
  - With JavaScript, **null** is for objects, **undefined** is for variables, properties, and methods
  - To be null, an object has to be defined, otherwise it will be undefined

```
if (typeof myObj !== "undefined" && myObj !== null)
```

JS Performance

- Reduce activity in Loops

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

- Reduce DOM Access

```
var obj;
obj = document.getElementById("demo");
obj.innerHTML = "Hello";
```

- Delay JS Loading
  - Putting your scripts at the bottom of the page body, lets the browser load the page first
  - While a script is downloading, the browser will not start any other downloads; in addition all parsing and rendering activity might be blocked

JS Performance

- If possible, you can add your script to the page by code, after the page has loaded

```
<script>
window.onload = downScripts;

function downScripts() {
    var element = document.createElement("script");
    element.src = "myScript.js";
    document.body.appendChild(element);
}
</script>
```

JS Forms

- HTML form validation can be done by JavaScript
- If a form field (fname) is empty, this function alerts a message, and returns false, to prevent the form from being submitted:

```
function validateForm() {
    var x = document.forms["myForm"]["fname"].value;
    if (x == "") {
        alert("Name must be filled out");
        return false;
    }
}
```

JS Forms

- The JS function can be called when the form is submitted:

```
<form name="myForm" action="demo_form.asp"
onsubmit="return validateForm()" method="post">
Name: <input type="text" name="fname">
<input type="submit" value="Submit">
</form>
```

JS Forms: Data Validation

- Data validation is the process of ensuring that user input is clean, correct, and useful
  - has the user filled in all required fields?
  - has the user entered a valid date?
  - has the user entered text in a numeric field?
- Validation can be defined by many different methods, and deployed in many different ways
- **Server side validation** is performed by a web server, after input has been sent to the server
- **Client side validation** is performed by a web browser, before input is sent to a web server

JS Forms API

- If an input field contains invalid data, display a message:

```
<input id="id1" type="number" min="100" max="300" required>
<button onclick="myFunction()">OK</button>

<p id="demo"></p>

<script>
function myFunction() {
    var inpObj = document.getElementById("id1");
    if (inpObj.checkValidity() == false) {
        document.getElementById("demo").innerHTML = inpObj.validationMessage;
    }
}
</script>
```

Validity Properties

Property	Description
customError	Set to true, if a custom validity message is set.
patternMismatch	Set to true, if an element's value does not match its pattern attribute.
rangeOverflow	Set to true, if an element's value is greater than its max attribute.
rangeUnderflow	Set to true, if an element's value is less than its min attribute.
stepMismatch	Set to true, if an element's value is invalid per its step attribute.
tooLong	Set to true, if an element's value exceeds its maxLength attribute.
typeMismatch	Set to true, if an element's value is invalid per its type attribute.
valueMissing	Set to true, if an element (with a required attribute) has no value.
valid	Set to true, if an element's value is valid.

Example

```
<input id="id1" type="number" max="100">
<button onclick="myFunction()">OK</button>

<p id="demo"></p>

<script>
function myFunction() {
    var txt = "";
    if (document.getElementById("id1").validity.rangeOverflow) {
        txt = "Value too large";
    }
    document.getElementById("demo").innerHTML = txt;
}
</script>
```