

Introduction to Modern JavaScript (ES6+)

Course Objectives

- After successfully completing this course, you will be able to:
 - Store information in variables and objects
 - Create code blocks for conditional logic and looping in code
 - Utilize browser developer tools for debugging JavaScript code
 - Create event handlers to respond to browser events
 - Create and call functions with parameters
 - Parse and create JSON objects
 - Create browser side scripts for manipulating the Document Object Model (DOM)
 - Create asynchronous calls to a Web server using AJAX
 - Discuss code organization techniques, including modules

Course Outline

- Chapter 0 Introduction to JavaScript Programming
- Chapter 1 Getting Started with JavaScript
- Chapter 2 JavaScript Syntax, Creating and Displaying Data
- Chapter 3 Code Blocks
- Chapter 4 Working with Built-In Objects and Arrays
- Chapter 5 Browser Object Model
- Chapter 6 More with the Document Object Model (DOM)
- Chapter 7 Using JavaScript with Forms
- Chapter 8 Object Oriented JavaScript
- Chapter 9 More Functions and Code Organization
- Chapter 10 JSON
- Chapter 11 Asynchronous Programming with JavaScript

This is a fast-paced customized course

- Normally, this is a 5 day course
- This course has been customized to focus on key concepts
- The course is heavily based upon demos
- Practice with the demos! Run them, change, them run again

Student Introductions

- Name
- Company / Department
- Current Position
- Background
- Expectations
- What do you like to do for fun?



About Your Instructor

- Judy Lipinski, PMP
 - judy@karmoxie.com
 - www.karmoxie.com
- Background:
 - Web & Mobile Development
 - Software Engineering, Consulting, Project Management
- Expectations:
 - That you learn something new and enjoy class

Prerequisites for this Course

- This is a hands-on programming class
- Attendees should have previous experience or working knowledge of developing software applications, as well as basic HTML, CSS, and JavaScript
- Real world programming experience is a must

Review: Basic HTML

```
<!doctype>
<html>
<head>
<title>A Sample HTML Doc</title>
</head>
<body>
<h3>A heading</h3> <!-- comment won't show up in browser view -->
<div>
    <p>Paragraphs are block-level elements</p>
    <p>Spans are <span="important">inline</span> elements</p>
    <p>Divs are block-level, and can contain other block-level<p>
</div>
</body>
</html>
```

Links

- A link can lead to....
 - a new page
 - a document or image, such as PDF
 - an internal section of a page

```
<a href="http://www.karmoxie.com/contact">  
  
</a>  
Visit our contact page for address and phone number.  
<a href="#moreinfo">View more info below</a>  
...  
<div id ="moreinfo"> Here is more info... </div>
```

Lists

- Are lists block-level or inline?

```
<p>Things I Like to Do</p>
<ol>
    <li>Wash Car</li>
    <li>Finish Taxes</li>
    <li class="favorite">Vote</li>
</ol>

<p>Things I DO NOT Like to Do</p>
<ul>
    <li>Wash Dishes</li>
    <li>Start Taxes</li>
    <li>Go to the DMV</li>
</ul>
```

Things I Like to Do

1. Wash Car
2. Finish Taxes
3. Vote

Things I DO NOT Like to Do

- Wash Dishes
- Start Taxes
- Go to the DMV

Replacing what we used to do in HTML

 no more bold 	--->	 this text is important
<i> no more italics </i>	--->	 emphasize text
<blink> blinking... </blink>	--->	never use blink!
<p> </p>	--->	p { color:red; font-size: 3px }

Chapter 1:

Getting Started with JavaScript

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Define JavaScript and ECMAScript
- Discuss available tools and editors
- Look at strategies for adding JS to webpages

Getting Started with JavaScript



What is JavaScript?

The ECMAScript Standard

Tools to help with JavaScript

Adding JavaScript to web pages

JavaScript ("JS")

- Invented by Brendan Eich in 1995 for Netscape Navigator
 - written in 10 days!
 - “LiveScript”
- A dynamic programming language
- Needs an engine
 - Websites provide engine and allow for dynamic behavior
 - Node.js provides environment for outside of browser

Examples of JavaScript Usage in Browser

- Carousels
- Image galleries
- Fluctuating layouts
- Responses to button clicks
- Dynamically create HTML
- Set CSS styles
- APIs built into web browsers
 - Form validation
 - 2D and 3D graphics
 - Collecting and manipulating a video stream from webcams
 - Location API
 - Storage

Example libraries and frameworks using JS

jQuery

dōjō

 React Native

 ANGULARJS
by Google


ionic

 ELECTRON




Visual Studio Code

Getting Started with JavaScript

History of JavaScript



The ECMAScript Standard

Tools to help with JavaScript

Strategies in adding JavaScript

ECMAScript IS JavaScript

- In 1996 Netscape submitted JS to ECMA International
- ECMAScript is the language specification
 - ECMAScript 3 - 1999 baseline for modern JS
 - Advances were held up by competing parties: MS, Yahoo, Adobe
- Open source and developer communities
 - Ajax - 2005, jQuery, Dojo, MooTools, Prototype
 - 2009 - Harmony - ECMAScript 5 released
- ES6 (ES2015) not supported by all browsers, supported in Node

ECMAScript Versions

- <https://en.wikipedia.org/wiki/ECMAScript#Versions>
- Which Browsers Support which Versions
 - <https://kangax.github.io/compat-table/es5/>
- We will be using Chrome
 - IE does not have the best support for ES6
- Can run processors to convert ES6 to ES5

ECMAScript defines:

- Language syntax
 - parsing rules, keywords, control flow, object literal initialization...
- Error handling mechanisms
 - throw, try/catch, ability to create user-defined Error types
- Types
 - boolean, number, string, function, object...
- A prototype-based inheritance mechanism
- Built-in objects and functions
 - JSON, Math, Array.prototype methods, Object introspection methods...
- Strict mode

Getting Started with JavaScript

History of JavaScript

The ECMAScript Standard



Tools to help with JavaScript

Strategies in adding JavaScript

Editors

- Any text editor can be used, popular choices include:
 - Atom
 - Brackets
 - WebStorm
 - \$100 commercial, \$50 personal, \$25 academic, free for open source developers upon application approval
 - Visual Studio Code (we will be using in this course)
- For quick shared examples online:
 - jsfiddle
 - codepen

Using Browser Developer tools and console.log()

- You can write to the developer tools console, using `console.log('message')`

`<script>`

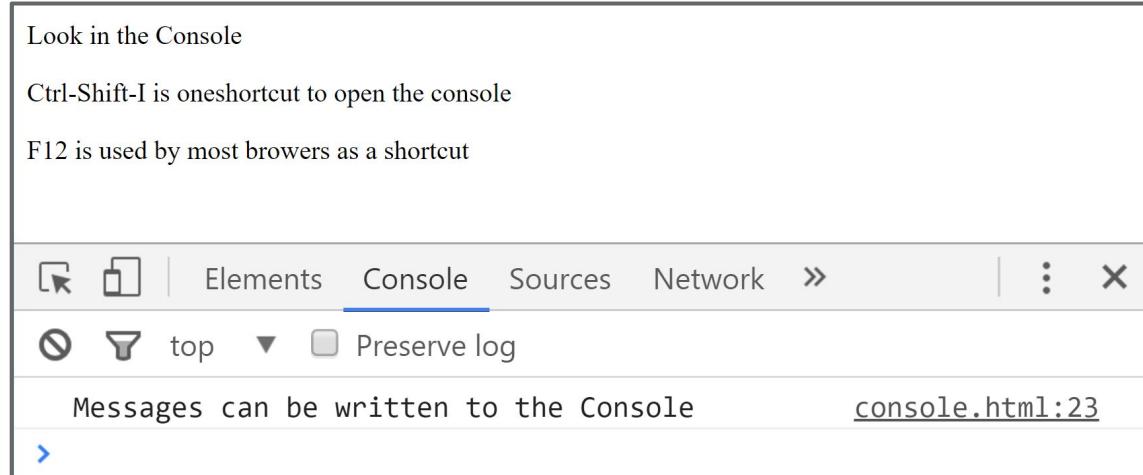
```
  console.log('Messages can be written to the Console');
```

`</script>`

Look in the Console

Ctrl-Shift-I is one shortcut to open the console

F12 is used by most browsers as a shortcut



Getting Started with JavaScript

History of JavaScript

The ECMAScript Standard

Tools to help with JavaScript



Strategies in adding JavaScript

JavaScript is added using the <script> tag

```
<!-- Prior to HTML5, needed to specify type attribute. -->
<script type="text/javascript">
    alert('I'm an annoying little pop-up!');
</script>

<!-- No longer need type. -->
<script>
    console.log('I am much nicer to see what is going on!');
</script>
```

JavaScript Files

- JavaScript is often put in separate files
 - It enables the code to be shared
 - It allows frameworks to be used
- There are often multiple script tags loading many files

5

```
<script src="..scripts/header.js"></script>
```

Where to put the <script> tag

- The location of script takes can have significant performance issues
 - Web pages often have a lot of JavaScript and need to load fast
- Script tags can go in the HTML <head>
 - Browsers will wait for the JavaScript to load before rendering HTML
- Script tags can go just before the </body> tag
 - Document is unresponsive until JavaScript loaded

Best practice: Load scripts with ‘defer’

- Without `defer` scripts are loaded, parsed and executed in the order in which they are encountered
 - HTML parsing pauses while this happens
- Using `defer`, HTML parsing continues while the scripts are being loaded
 - It delays parsing JavaScript files until all HTML has been parsed
- `defer` can only be used with the `src` attribute
- About 80% of browsers support `defer`

```
<script src="someScript.js" defer="defer"></script>
```

Starting a function from a browser event

- An **inline** call to a function, after the page has loaded...

```
</head>
...
<script>
    function bodyOnloadFunction() {
        document.getElementById('targetedH2').innerHTML = "New text for h2";
    }
</script>
</head>
<body onload="bodyOnloadFunction()">
    <h2 id="targetedH2">
        In the h2
    </h2>
    ...
</body>
```

Chapter Summary

In this chapter we have:

- Defined JavaScript ad ECMAScript
- Discussed available tools and editors
- Looked at strategies for adding JS to webpages

Chapter 2: JavaScript Syntax, Creating and Displaying Data

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Discuss basics of JS syntax
- Review the 6 primitive JS types
- Practice with operators
- Create and access arrays
- Create and access object data
- Create and call named functions with and without parameters
- Add and invoke object methods

JavaScript Syntax, Creating and Displaying Data



Statements, Variables and Types

Displaying JavaScript data and debugging

Operating on Primitives

Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Multiple and Single Line Comments

```
18  <script>
19    /*
20      Everything in between the stars and
21      slashes is a comment.
22    */
23    var x = 3;
24    const shortPi = 3.14; //i wrote this on Pi day!
25    let y = 10;
26  </script>
```

Best practice: End Statements with a semicolon

- Declaring variables is an example of a statement
- In JavaScript, a semicolon is automatically inserted when:
 - two statements are separated by a line terminator
 - two statements are separated by a closing brace ('}'')
 - a line terminator follows a break, continue, return, or throw
- Best practice: **do not rely on automatic inserts!**

```
var x = 3;  
const shortPi = 3.14;  
let y = 10;
```

About Variables

- They are containers to store values
- JS naming IS case sensitive
 - myVar != MyVar
- Can check for valid names here: <https://mothereff.in/js-variables>

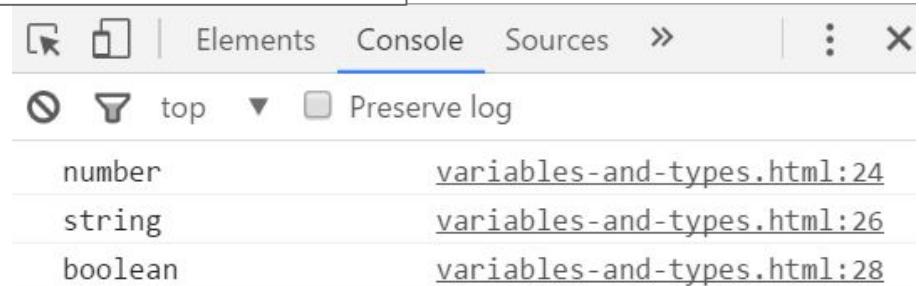
Declaring Variables with `let`, `const` or `var`

- `var`
 - the way we used to do things in JS
 - will still see this a LOT
- As of ES6 `const` and `let` are preferred, indicates intention:
 - `const` variables cannot be re-assigned
 - `let` variables can be re-assigned
- It is possible to declare without `var`, `const`, `let`
 - **but don't! Pollutes the global namespace**

JavaScript is loosely typed aka it uses dynamic typing

- No type declaration, determined automatically
- Same variable can be used with different types:

```
24 | let x = 3;  console.log(typeof x);
25 |
26 | x = 'abc';  console.log(typeof x);
27 |
28 | x = true;   console.log(typeof x);
```



The screenshot shows the Chrome DevTools Console tab. The code from the previous block is run sequentially, and the results are displayed below. The first three lines output 'number' because they log the primitive values 3, 'abc', and true respectively. The fourth line outputs 'boolean' because it logs the reference to the variable x, which now points to the boolean value true.

Output	Line Number
number	variables-and-types.html:24
string	variables-and-types.html:26
boolean	variables-and-types.html:28

7 Types in ECMAScript

- 6 data types that are primitives (and immutable):
 - number
 - string
 - boolean
 - undefined
 - null
 - Symbol (new in ECMAScript 6)
- Object

Only one number type in JavaScript

- The double-precision 64-bit binary format IEEE 754
- No specific type for integers
- Has 3 symbolic values: +Infinity, -Infinity, and NaN (not-a-number)

```
33 | let z = 3.13; console.log(typeof z);
34 | let divideByZero = (z/+0); console.log(divideByZero);
35 | divideByZero = (z/-0); console.log(divideByZero);
```

Numbers, min and max

- Number.MAX_VALUE
 - the maximum numeric value
- Number.MIN_VALUE
 - the smallest positive value (>0)
- Number.MAX_SAFE_INTEGER
 - the maximum safe integer ($2^{53} - 1$)
- Number.MIN_SAFE_INTEGER
 - the maximum safe integer ($-(2^{53} - 1)$)

undefined and null

- **undefined** - the value has not yet been assigned
- **null** - absence of object, set explicitly
 - `typeof` returns 'object'
 - ECMAScript proposal for returning 'null' for `typeOf` was rejected

```
30 | let y;    console.log(typeof y);
31 | y = null; console.log(typeof y);
```

undefined

[variables-and-types.html:30](#)

object

[variables-and-types.html:31](#)

String literals in quotes

- Literals can be represented with single quotes or double quotes

```
let dblQuoteString = "Isn't it nice that I can contain single quotes";  
let singleQuoteString = 'Well, you may "think" that\'s cool...';  
console.log(dblQuoteString);  
console.log(singleQuoteString);
```

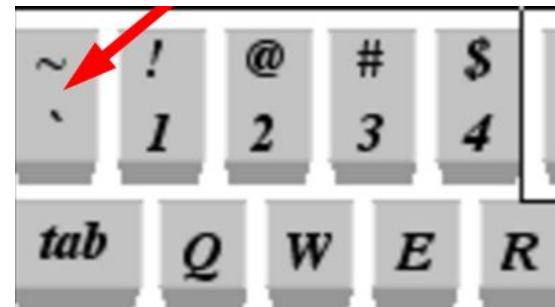
Using ES6 backticks

- You can use variables within strings by using backticks

```
let firstName = 'John';  
let lastName = 'Herilla';
```

//note these are not apostrophies...but back ticks

```
alert(`  
    ${firstName} ${lastName}  
`);
```



String template literals with backticks

- Can contain placeholders with \${expression}
- Can contain quotes

```
41 let name="Remembrance O'Neal";
42 let myString = `      'Single' and "Double" quotes work
43           Backticks allow other variables to be referenced: name= ${name}
44           Line ${2+1}  Expressions can be evaluated
45 `;
46 console.log(myString);
```

'Single' and "Double" quotes work

Backticks allow other variables to be referenced: name= Remembrance O'Neal
Line 3 Expressions can be evaluated

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types



Displaying JavaScript data and debugging

Operating on Primitives

Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Grouping messages in console.log()

```
19      var x = 3;
20      const shortPi = 3.14;
21      let y = 10;
22
23      let z = x + y;
24
25      let myString = "abc";
26
27      console.group('Plus symbol... ');
28      console.log('x = ' + x);
29      console.log('x+1 = ' + ++x);
30      console.log(z + myString);
31      console.groupEnd();
```

- Group messages using console.group

The DOM can be used to update the browser content

- The document object has multiple methods, for now:
 - `let targetId = document.getElementById("targetedId");`
 - `targetId.textContent = "Replaced Content";`

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

```
let targetId = document.getElementById("targetedId");
targetId.innerHTML = "Replaced Content";
```

Use .innerHTML for html content

- The document object has multiple methods, for now:
 - `document.getElementById("targetedId");`
 - `targetId.textContent = "Replaced Content";`

```
let targetId = document.getElementById("targetedId");
targetId.innerHTML = "<p>Replaced HTML Based Content</p>";
```

Can use document.write to write to page

```
28 <script>
29     document.write("What is 2+4? :");
30     document.write(2 + 4);
31     document.write('<br />And 3 + 1?<br /> ');
32     document.write(3+1);
33 </script>
```

Displaying Data

[Back to Demos index](#)

Replaced Content

What is 2+4? :6
And 3 + 1?
4

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types

Displaying JavaScript data and debugging



Operating on Primitives

Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Arithmetic Operators

Assignment operator	=	var myVariable = 'Bob';
Addition or Concatenation	+	6 + 9; //15 "Hello " + "world!"; // Hello world!
Pre Increment/Decrement	++x --z	x = 3; y = ++x; console.log(y) // 4
Post Increment/Decrement	x++ z--	x = 3;y = x++; console.log(y) // 3
subtract, multiply, divide	- , *, /	9 - 3; // 6 8 * 2; // 16 9 / 3; // 3
Remainder/Modulus	%	x=12 console.log(x%3) // 0 console.log (x%5) // 2
Exponent	**	8**2 // 64

Equality and Identity Operators

Equality Operator	<code>==</code>	check and allow conversion <code>a = '1'</code> <code>b = 1</code> <code>(a==b) → true</code>
Strict equal, checks type (<i>preferred</i>)	<code>===</code>	<code>(a === b) → false</code>
Negation, not equal	<code>!=</code>	<code>var myVariable = 3;</code> <code>myVariable != 4;</code>
Strict not equal true if same type but not equal, or if different types	<code>!==</code>	<code>3 !== '3'</code>

Relational Operators

Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=
Logical AND	&&
Logical OR	

Warning: => is not an operator but is used with Arrow Functions

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types

Displaying JavaScript data and debugging

Operating on Primitives



Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Creating and Accessing Arrays

- Arrays can be created with square brackets or using `new Array()`

```
const simpleNameArray = ['Adam', 'Judy', 'Cody'];
const ages = new Array(45, 41, 1);
```

- Elements of the array can be accessed by the index
 - starting at 0

```
console.log(simpleNameArray[0] + ' is ' +
            ages[0] + ' years old');
```

Finding length of array

- Arrays have a length to show the number of elements

```
11      const numbers = new Array( 1, 2, 3, 4, 5 ) ;
12      console.log( 'Numbers length ' + numbers.length ) ;
```

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types

Displaying JavaScript data and debugging

Operating on Primitives

Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Objects

- You can group together multiple values in an Object
- Object literals contain key-value pairs
- Advantages over arrays to keep related information together
- Very helpful to retrieve properties of similar Objects
- Like "dictionaries" or "maps" in other languages
 - an unordered collection of key-value pairs

Create Object Literals or use new Object

- Here an Object is created as an Object Literal - using curly braces

```
var band1 = {  
    "name" : "Pink Floyd",  
    "country" : "England",  
    "yearFormed" : 1965,  
    "genres" : ["Progressive rock", "psychedelic rock", "art rock"]  
}
```

- This is the equivalent of:

```
var band2 = new Object();  
band2.name = "Pink Floyd";  
band2.country = "England";  
band2.yearFormed = 1965;  
band2.genres = ["Progressive rock", "psychedelic rock", "art rock"];
```

Referencing Properties of Objects

- Properties can be referenced using dot notation or bracket notation

```
//access properties using dot notation or □  
console.log("country 1 = " + band1.country);  
console.log("country 2 = " + band2["country"]);
```

Arrays in Objects

- Object literals can contain array literals:

```
var band1 = {  
    "name" : "Pink Floyd",  
    "country" : "England",  
    "yearFormed" : 1965,  
    "genres" : ["Progressive rock", "psychedelic rock", "art rock"]  
}
```

Objects in Arrays

- Array literals can contain object literals:

```
//can have arrays of objects
const peopleArray = [
    { name: 'Adam', age=45 },
    { name: 'Judy', age=41 },
    { name: 'Cody', age=1 },
];
```

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types

Displaying JavaScript data and debugging

Operating on Primitives

Creating and Accessing Arrays

Creating Objects



Basic Functions & Methods

Functions perform tasks

- Function: a block of code designed to perform a particular task

```
function printStars() {  
    console.log('*****');  
}
```

- A JavaScript function is executed when "something" invokes it (calls it)
 - Call a function by using parenthesis

```
printStars();
```

Functions can take parameters

- Inside of the function the parameters behave as local variables

```
25  function printFullName(localFirstName, localLastName) {  
26      localFirstName = "Dr. " + localFirstName;  
27      console.log(localFirstName + ' ' + localLastName);  
28  }  
29  
30  firstName = 'Bob';  
31  lastName = 'Williams';  
32  printFullName(firstName, lastName);  
33  console.log('after function call, name is ' + firstName);
```

Dr. Bob Williams

after function call, name is Bob

Functions can return a value

- The returned value can be stored or used in an expression

```
36  function returnFullName(firstName, lastName) {  
37      return (firstName + ' ' + lastName);  
38  }  
39  
40  const fullName = returnFullName('Jackie', 'Chan');  
41  
42  console.log(fullName);
```

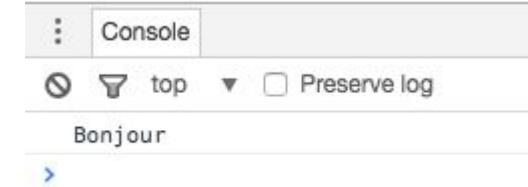
Function scope and conflicts

- If two scripts are included function names may conflict
- The last one loaded replaces the others

```
5      <script>
6          function getText() {
7              return "Hello" ;
8          }
9      </script>
10     <script>
11         function getText() {
12             return "Bonjour" ;
13         }
14     </script>
```

```
21
22
23
```

```
<script>
    console.log( getText() ) ;
</script>
```



Methods are functions in objects

- A method is simply a key with a function as a value
- A method is called by the key with parentheses

```
15      const Person = {  
16          "firstName": "Carole",  
17          "lastName": "King",  
18          "fullName": function() {  
19              return this.firstName + ' ' + this.lastName ;  
20          }  
21      }
```

32 console.log("Person " + Person.fullName()) ;

Can use default parameters in functions

```
34          function multiply1( a, b ) {  
35              b = typeof b !== 'undefined' ? b : 2 ;  
36              return a * b ;  
37          }  
38  
39          console.log( multiply1( 5 ) ) ;  
  
41          function multiply2( a, b = 2 ) {  
42              return a * b ;  
43          }  
44  
45          console.log( multiply2( 5 ) ) ;
```

Chapter Summary

In this chapter we have:

- Discussed basic of JS syntax
- Reviewed the 6 primitive JS types
- Practiced with operators
- Worked with arrays
- Created and accessed object data

Chapter 2: JavaScript Syntax, Creating and Displaying Data

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Discuss basics of JS syntax
- Review the 6 primitive JS types
- Practice with operators
- Create and access arrays
- Create and access object data
- Create and call named functions with and without parameters
- Add and invoke object methods

JavaScript Syntax, Creating and Displaying Data



Statements, Variables and Types

Displaying JavaScript data and debugging

Operating on Primitives

Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Multiple and Single Line Comments

```
18  <script>
19    /*
20      Everything in between the stars and
21      slashes is a comment.
22    */
23    var x = 3;
24    const shortPi = 3.14; //i wrote this on Pi day!
25    let y = 10;
26  </script>
```

Best practice: End Statements with a semicolon

- Declaring variables is an example of a statement
- In JavaScript, a semicolon is automatically inserted when:
 - two statements are separated by a line terminator
 - two statements are separated by a closing brace ('}'')
 - a line terminator follows a break, continue, return, or throw
- Best practice: **do not rely on automatic inserts!**

```
var x = 3;  
const shortPi = 3.14;  
let y = 10;
```

About Variables

- They are containers to store values
- JS naming IS case sensitive
 - myVar != MyVar
- Can check for valid names here: <https://mothereff.in/js-variables>

Declaring Variables with `let`, `const` or `var`

- `var`
 - the way we used to do things in JS
 - will still see this a LOT
- As of ES6 `const` and `let` are preferred, indicates intention:
 - `const` variables cannot be re-assigned
 - `let` variables can be re-assigned
- It is possible to declare without `var`, `const`, `let`
 - **but don't! Pollutes the global namespace**

JavaScript is loosely typed aka it uses dynamic typing

- No type declaration, determined automatically
- Same variable can be used with different types:

```
24 | let x = 3;  console.log(typeof x);
25 |
26 | x = 'abc';  console.log(typeof x);
27 |
28 | x = true;   console.log(typeof x);
```



The screenshot shows the Chrome DevTools Console tab. The code from the previous block is run sequentially, and the results are displayed below. The first three lines output 'number' because they log the primitive values 3, 'abc', and true respectively. The fourth line outputs 'boolean' because it logs the reference to the variable x, which now points to the boolean value true.

Output	Line Number
number	variables-and-types.html:24
string	variables-and-types.html:26
boolean	variables-and-types.html:28

7 Types in ECMAScript

- 6 data types that are primitives (and immutable):
 - number
 - string
 - boolean
 - undefined
 - null
 - Symbol (new in ECMAScript 6)
- Object

Only one number type in JavaScript

- The double-precision 64-bit binary format IEEE 754
- No specific type for integers
- Has 3 symbolic values: +Infinity, -Infinity, and NaN (not-a-number)

```
33 | let z = 3.13; console.log(typeof z);
34 | let divideByZero = (z/+0); console.log(divideByZero);
35 | divideByZero = (z/-0); console.log(divideByZero);
```

Numbers, min and max

- Number.MAX_VALUE
 - the maximum numeric value
- Number.MIN_VALUE
 - the smallest positive value (>0)
- Number.MAX_SAFE_INTEGER
 - the maximum safe integer ($2^{53} - 1$)
- Number.MIN_SAFE_INTEGER
 - the maximum safe integer ($-(2^{53} - 1)$)

undefined and null

- **undefined** - the value has not yet been assigned
- **null** - absence of object, set explicitly
 - `typeof` returns 'object'
 - ECMAScript proposal for returning 'null' for `typeOf` was rejected

```
30 | let y;    console.log(typeof y);
31 | y = null; console.log(typeof y);
```

undefined

[variables-and-types.html:30](#)

object

[variables-and-types.html:31](#)

String literals in quotes

- Literals can be represented with single quotes or double quotes

```
let dblQuoteString = "Isn't it nice that I can contain single quotes";  
let singleQuoteString = 'Well, you may "think" that\'s cool...';  
console.log(dblQuoteString);  
console.log(singleQuoteString);
```

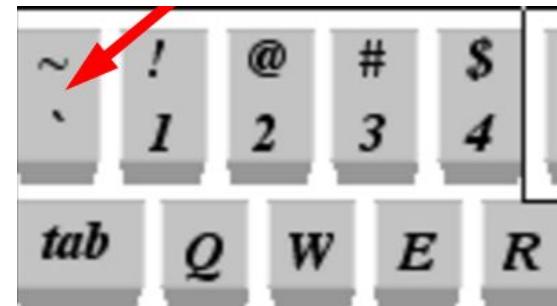
Using ES6 backticks

- You can use variables within strings by using backticks

```
let firstName = 'John';
let lastName = 'Herilla';
```

//note these are not apostrophies...but back ticks

```
alert(`  
    ${firstName} ${lastName}  
`);
```



String template literals with backticks

- Can contain placeholders with \${expression}
- Can contain quotes

```
41 let name="Remembrance O'Neal";
42 let myString = `      'Single' and "Double" quotes work
43           Backticks allow other variables to be referenced: name= ${name}
44           Line ${2+1}  Expressions can be evaluated
45 `;
46 console.log(myString);
```

'Single' and "Double" quotes work

Backticks allow other variables to be referenced: name= Remembrance O'Neal
Line 3 Expressions can be evaluated

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types



Displaying JavaScript data and debugging

Operating on Primitives

Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Grouping messages in console.log()

```
19      var x = 3;
20      const shortPi = 3.14;
21      let y = 10;
22
23      let z = x + y;
24
25      let myString = "abc";
26
27      console.group('Plus symbol... ');
28      console.log('x = ' + x);
29      console.log('x+1 = ' + ++x);
30      console.log(z + myString);
31      console.groupEnd();
```

- Group messages using console.group

The DOM can be used to update the browser content

- The document object has multiple methods, for now:
 - `let targetId = document.getElementById("targetedId");`
 - `targetId.textContent = "Replaced Content";`

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

```
let targetId = document.getElementById("targetedId");
targetId.innerHTML = "Replaced Content";
```

Use .innerHTML for html content

- The document object has multiple methods, for now:
 - `document.getElementById("targetedId");`
 - `targetId.textContent = "Replaced Content";`

```
let targetId = document.getElementById("targetedId");
targetId.innerHTML = "<p>Replaced HTML Based Content</p>";
```

Can use document.write to write to page

```
28 <script>
29     document.write("What is 2+4? :");
30     document.write(2 + 4);
31     document.write('<br />And 3 + 1?<br /> ');
32     document.write(3+1);
33 </script>
```

Displaying Data

[Back to Demos index](#)

Replaced Content

What is 2+4? :6
And 3 + 1?
4

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types

Displaying JavaScript data and debugging



Operating on Primitives

Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Arithmetic Operators

Assignment operator	=	var myVariable = 'Bob';
Addition or Concatenation	+	6 + 9; //15 "Hello " + "world!"; // Hello world!
Pre Increment/Decrement	++x --z	x = 3; y = ++x; console.log(y) // 4
Post Increment/Decrement	x++ z--	x = 3;y = x++; console.log(y) // 3
subtract, multiply, divide	- , *, /	9 - 3; // 6 8 * 2; // 16 9 / 3; // 3
Remainder/Modulus	%	x=12 console.log(x%3) // 0 console.log (x%5) // 2
Exponent	**	8**2 // 64

Equality and Identity Operators

Equality Operator	<code>==</code>	check and allow conversion <code>a = '1'</code> <code>b = 1</code> <code>(a==b) → true</code>
Strict equal, checks type (<i>preferred</i>)	<code>===</code>	<code>(a===b) → false</code>
Negation, not equal	<code>!=</code>	<code>var myVariable = 3;</code> <code>myVariable != 4;</code>
Strict not equal true if same type but not equal, or if different types	<code>!==</code>	<code>3 !== '3'</code>

Relational Operators

Less than	<
Greater than	>
Less than or equal	<=
Greater than or equal	>=
Logical AND	&&
Logical OR	

Warning: => is not an operator but is used with Arrow Functions

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types

Displaying JavaScript data and debugging

Operating on Primitives



Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Creating and Accessing Arrays

- Arrays can be created with square brackets or using `new Array()`

```
const simpleNameArray = ['Adam', 'Judy', 'Cody'];
const ages = new Array(45, 41, 1);
```

- Elements of the array can be accessed by the index
 - starting at 0

```
console.log(simpleNameArray[0] + ' is ' +
            ages[0] + ' years old');
```

Finding length of array

- Arrays have a length to show the number of elements

```
11      const numbers = new Array( 1, 2, 3, 4, 5 ) ;
12      console.log( 'Numbers length ' + numbers.length ) ;
```

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types

Displaying JavaScript data and debugging

Operating on Primitives

Creating and Accessing Arrays

Creating Objects

Basic Functions & Methods

Objects

- You can group together multiple values in an Object
- Object literals contain key-value pairs
- Advantages over arrays to keep related information together
- Very helpful to retrieve properties of similar Objects
- Like "dictionaries" or "maps" in other languages
 - an unordered collection of key-value pairs

Create Object Literals or use new Object

- Here an Object is created as an Object Literal - using curly braces

```
var band1 = {  
    "name" : "Pink Floyd",  
    "country" : "England",  
    "yearFormed" : 1965,  
    "genres" : ["Progressive rock", "psychedelic rock", "art rock"]  
}
```

- This is the equivalent of:

```
var band2 = new Object();  
band2.name = "Pink Floyd";  
band2.country = "England";  
band2.yearFormed = 1965;  
band2.genres = ["Progressive rock", "psychedelic rock", "art rock"];
```

Referencing Properties of Objects

- Properties can be referenced using dot notation or bracket notation

```
//access properties using dot notation or □  
console.log("country 1 = " + band1.country);  
console.log("country 2 = " + band2["country"]);
```

Arrays in Objects

- Object literals can contain array literals:

```
var band1 = {  
    "name" : "Pink Floyd",  
    "country" : "England",  
    "yearFormed" : 1965,  
    "genres" : ["Progressive rock", "psychedelic rock", "art rock"]  
}
```

Objects in Arrays

- Array literals can contain object literals:

```
//can have arrays of objects
const peopleArray = [
    { name: 'Adam', age=45 },
    { name: 'Judy', age=41 },
    { name: 'Cody', age=1 },
];
```

JavaScript Syntax, Creating and Displaying Data

Statements, Variables and Types

Displaying JavaScript data and debugging

Operating on Primitives

Creating and Accessing Arrays

Creating Objects



Basic Functions & Methods

Functions perform tasks

- Function: a block of code designed to perform a particular task

```
function printStars() {  
    console.log('*****');  
}
```

- A JavaScript function is executed when "something" invokes it (calls it)
 - Call a function by using parenthesis

```
printStars();
```

Functions can take parameters

- Inside of the function the parameters behave as local variables

```
25  function printFullName(localFirstName, localLastName) {  
26      localFirstName = "Dr. " + localFirstName;  
27      console.log(localFirstName + ' ' + localLastName);  
28  }  
29  
30  firstName = 'Bob';  
31  lastName = 'Williams';  
32  printFullName(firstName, lastName);  
33  console.log('after function call, name is ' + firstName);
```

Dr. Bob Williams

after function call, name is Bob

Functions can return a value

- The returned value can be stored or used in an expression

```
36  function returnFullName(firstName, lastName) {  
37      return (firstName + ' ' + lastName);  
38  }  
39  
40  const fullName = returnFullName('Jackie', 'Chan');  
41  
42  console.log(fullName);
```

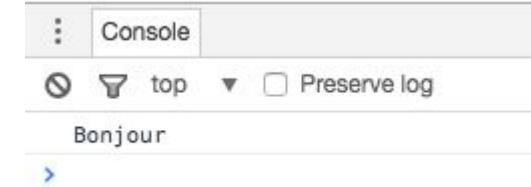
Function scope and conflicts

- If two scripts are included function names may conflict
- The last one loaded replaces the others

```
5      <script>
6          function getText() {
7              return "Hello" ;
8          }
9      </script>
10     <script>
11         function getText() {
12             return "Bonjour" ;
13         }
14     </script>
```

```
21
22
23
```

```
<script>
    console.log( getText() ) ;
</script>
```



Methods are functions in objects

- A method is simply a key with a function as a value
- A method is called by the key with parentheses

```
15      const Person = {  
16          "firstName": "Carole",  
17          "lastName": "King",  
18          "fullName": function() {  
19              return this.firstName + ' ' + this.lastName ;  
20          }  
21      }
```

```
32           console.log( "Person " + Person.fullName() ) ;
```

Can use default parameters in functions

```
34          function multiply1( a, b ) {  
35              b = typeof b !== 'undefined' ? b : 2 ;  
36              return a * b ;  
37          }  
38  
39          console.log( multiply1( 5 ) ) ;  
  
41          function multiply2( a, b = 2 ) {  
42              return a * b ;  
43          }  
44  
45          console.log( multiply2( 5 ) ) ;
```

Chapter Summary

In this chapter we have:

- Discussed basic of JS syntax
- Reviewed the 6 primitive JS types
- Practiced with operators
- Worked with arrays
- Created and accessed object data

Chapter 3:

Code Blocks

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Create if, else and switch statements
- Construct different types of for loops
- Use keywords to change flow in loops
- Write code to handle errors

Code Blocks



Blocks and Scope

Using Conditional Logic

Looping

Error Handling

Blocks

- A block is a section of code in braces {}
 - Blocks are usually associated with control structure such as if and for
- Variables created inside a block with let or const have local scope
 - They can't be accessed outside of the block

```
11      {  
12          var name3 = 'Clare' ;  
13          let name4 = 'Danny' ;  
14          console.log( 'Local ' + name3 ) ;  
15          console.log( 'Local ' + name4 ) ;  
16      }  
17      console.log( 'Global ' + name3 ) ; Local Clare  
18      console.log( 'Global ' + name4 ) ; Local Danny  
                                Global Clare  
✖ ► Uncaught ReferenceError: name4 is not defined  
at scope.html:18
```

Scope

- Variables declared outside of a block or function have global scope
 - They can be accessed from anywhere
 - In a browser, they are added to window object
- Variables created without var, let or const are global
 - Can prevent this by adding "use strict;" to the script

```
6          var name1 = 'Alice' ;
7          let name2 = 'Bob' ;
8          console.log( 'Global ' + name1 ) ;
9          console.log( 'Global ' + name2 ) ;
```

Functions and Scope

- Variables declared inside a function have local scope
 - They can only be accessed from inside the function or inner function
 - This also includes variables declared with var

```
20      function printName1() {  
21          var name5 = 'Emma' ;  
22          let name6 = 'Frank' ;  
23          function printName2() {  
24              var name7 = 'Greta' ;  
25              let name8 = 'Harold' ;  
26              console.log( 'Inner ' + name5 ) ;  
27              console.log( 'Inner ' + name6 ) ;  
28          }  
29          printName2() ;  
30          console.log( 'Inner ' + name7 ) ;  
31          console.log( 'Inner ' + name8 ) ;  
32      }  
33      printName1() ;
```

Inner Emma
Inner Frank

✖ ► Uncaught ReferenceError: name7 is not defined
at printName1 (scope.html:30)
at scope.html:33

Code Blocks

Blocks and Scope

➤ **Using Conditional Logic**

Looping

Error Handling

Using if, else if, else

- Use boolean checks to execute blocks of code

```
21 //if, else if
22 if (hour > 12 && hour < 13) {
23     console.log('It is lunch time');
24 }
25 else if (hour > 8 && hour < 17) {
26     console.log('class is in session');
27 }
28 else {
29     console.log('go home - class is not in session!');
30 }
```

Revisiting boolean values: Truth-y and False-y

- Expressions can be used in a boolean context to return true or false
- Most values are true - truthy
- The following are always false - falsy
 - The keyword `false`
 - The number `0`
 - The empty string `""` or `' '`
 - The keywords `null` and `undefined`
 - The special number Not-a-Number `NaN`

Comparing Falsy Values

- Falsy values have unusual comparison rules
- Falsy values false, 0 and "" are equivalent $0 == \text{false}$
- null and undefined are only equivalent to themselves $\text{null} == \text{null}$
- NaN is not equivalent to anything including itself

switch

- Match expressions, then execute code until a break

```
47  switch (new Date().getDay()) {  
48      case 0:  
49      case 6:  
50          text = "We are closed today.";  
51          break;  
52      default:  
53          text = "9am to 5pm";  
54  }  
55  
56  document.write('<br />Store hours today: ' + text);
```

Code Blocks

Blocks and Scope

Using Conditional Logic



Looping

Error Handling

Types of for loops

- **for** - loops through a block of code a number of times
- **for/in** - loops through the properties of an object
- **while** - loops through a block of code while a specified condition is true
- **do/while** - also loops through a block of code while a specified condition is true

While Loops

- Check a condition, loop while it is true
- The loop executes zero or more times

```
19  while (counter <= 5) {  
20      console.log("Counter: " + counter++);  
21  }
```

Counter: 0
Counter: 1
Counter: 2
Counter: 3
Counter: 4
Counter: 5

Do while loops

- A do-while loop executes one or more times

```
7      let counter = 0 ;
8      do {
9          console.log( "Counter: " + counter++ ) ;
10     } while ( counter <= 5 ) ;
```

For loops execute a block of code a number of times

- Write it in 3 parts, separated by semicolons
 - a. initialization, runs once before code block
 - b. condition, checked each time before code block
 - c. way to change variable, executed after code block

A

B

C

```
19  for (let index = 0; index <= 5; index++) {  
20      console.log("At index: " + index);  
21  }
```

At index: 0

At index: 1

At index: 2

At index: 3

At index: 4

At index: 5

For loops commonly used with arrays

- Use length to set the condition check

```
19 const pets = [
20   {name: 'Birdy', type: 'cat'},
21   {name: 'Roxy', type: 'dog'}
22 ]
23
24 for (let index = 0; index < pets.length; index++) {
25   console.log(index);
26   console.log(pets[index].name + " is a " + pets[index].type);
27 }
```

For..In usage with objects

- Can retrieve object properties with for...in

```
19  const student =
20    {name: 'Sarah Wiessman',
21     email: 'sarah@email.com',
22     street: '123 Main St.',
23     city: 'Pittsburgh'
24   };
25
26
27 for (let prop in student) {
28   console.log('student.' + prop, '=' , student[prop]);
29 }
```

for..of usage with iterables

- The for...of loop iterates over array values and the letters of a string

```
11      let numbers = [1, 3, 5, 7, 9] ;
12      for ( const value of numbers ) {
13          console.log( value ) ;
14      }
15
16      let text = "Hello World" ;
17      for ( const letter of text ) {
18          console.log( letter ) ;
19      }
```

Exiting Loops with break

- The break statement causes the loop to terminate
- Always used in an if statement to terminate when something is found

```
16      let text = "Hello World" ;
17      for ( const letter of text ) {
18          if ( letter == 'o' ) {
19              break ;
20          }
21          console.log( letter ) ;
22      }
```

Skipping iterations with continue

- The continue statement terminates the current iteration and moves to the next
- Used in an if statement to skip unwanted values

```
16      let text = "Hello World" ;
17      for ( const letter of text ) {
18          if ( letter == 'o' ) {
19              continue ;
20          }
21          console.log( letter ) ;
22      }
```

Code Blocks

Blocks and Scope

Using Conditional Logic

Looping



Error Handling

Error Handling in JavaScript

- It is possible to handle errors in JavaScript

```
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
}
```

Throw Custom Errors

- You can throw an exception (throw an error)
- The exception can be a JavaScript string, a number, boolean or object:

```
throw "Too big"; // throw a text  
throw 500;      // throw a number
```

Error objects provide details

- You can handle errors with a try catch block

```
29  try {  
30      console.log(undefinedVar);  
31  }  
32  catch(err) {  
33      console.log(err);  
34  }
```

```
ReferenceError: undefinedVar is not defined  
at try-catch-finally.html:30
```

Chapter Summary

In this chapter we have:

- Created if, else and switch statements
- Constructed different types of for loops
- Used keywords to change flow in loops
- Wrote code to handle errors

Chapter 4: **Working with Built-In Objects** **and Arrays**

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Practice with built-in objects and methods
- Use Number Methods
- Create random values
- Create and display JavaScript Dates
- Work with Arrays

Working with Built-In Objects and Arrays



Built-In Objects

Using String Properties and Methods

Using Number Methods

The JavaScript Math object and Random

Creating and displaying JavaScript Dates

Practice with Arrays

Built-In Objects

- JavaScript has a number of built in objects
 - Values
 - Functions
 - Numbers and dates
 - Strings
 - Collections

Primitive wrapper objects in JavaScript

- Except for null and undefined, all primitive values have object equivalents
 - String for the string primitive
 - Number for the number primitive
 - Boolean for the boolean primitive
 - Symbol for the Symbol primitive
- These wrap around primitives and provide functionality

Working with Built-In Objects and Arrays

Built-In Objects

➤ **Using String Properties and Methods**

Using Number Methods

The JavaScript Math object and Random

Creating and displaying JavaScript Dates

Practice with Arrays

String is an array of letters

- Length of String

```
const alphabet = 'abcdefghijklmnopqrstuvwxyz';
console.log('Length is ' + alphabet.length);
```

- Can use for...of to iterate over a String

```
for (let letter of alphabet) {
    document.write(letter);
}
```

Searching within a String

- Find the position of a character or string

```
const str = "Which letter is j?";  
let pos = alphabet.indexOf("j");  
console.log('j is the ' + (pos+1) + 'th letter');
```

Breaking a string with a delimiter

- The `split()` method can break on a delimiter

```
var sentence = "Want these words as an array";
var wordArray = sentence.split(" "); //split on which character?
console.log(wordArray);
```

▶ ["Want", "these", "words", "as", "an", "array?"]

Slicing and returning a substring

- There are 3 methods for extracting a part of a string:
 - `slice(start, end)`
 - `substring(start, end)`
 - `substr(start, length)`

Using slice()

- Returns a new string, the end index is NOT inclusive

```
let longSentence = 'There she goes...';
let shorterSentence = longSentence.slice(0, 2);
console.log(longSentence);
console.log(shorterSentence);
```

There she goes...

Th

Using slice() with no end and negative indexes

- If slice is called with no value it returns the whole string
- A negative index counts from the end of the string

```
24      text = "The morning is upon us" ;
25      console.log( text.slice() ) ;
26      console.log( text.slice( -4 ) ) ;
27      console.log( text.slice( -4, -1 ) ) ;
28      console.log( text.slice( 0, -3 ) ) ;
```

The morning is upon us

n us

n u

The morning is upon

Using `substring(start, end)`

`string.substr(start, length)`

```
const spell = "Abracadabra!";
let substringSpell = spell.substring(1, 4);
console.log('substrSpell=' + substringSpell);
```

substrSpell=bra

Using substr(start, length)

string.substr(start, length)

```
const spell = "Abracadabra!";
let substrSpell = spell.substr(1, 4);
console.log('substrSpell=' + substrSpell);
```

substrSpell=brac

More String methods

```
48 const baseString = ' Just Some Text With SSpaces '
49 console.log('baseString.toUpperCase: ' + baseString.toUpperCase());
50 console.log('baseString.toLowerCase: ' + baseString.toLowerCase());
51 console.log('baseString.trim: ' + baseString.trim());
52 console.log("baseString.replace('SS','S')" + baseString.replace('SS','S'));
```

```
baseString.toUpperCase: JUST SOME TEXT WITH SSPACES
baseString.toLowerCase: just some text with sspaces
baseString.trim: Just Some Text With SSpaces
baseString.replace('SS','S') Just Some Text With Spaces
```

Working with Built-In Objects and Arrays

Built-In Objects

Using String Properties and Methods

► **Using Number Methods**

The JavaScript Math object and Random

Creating and displaying JavaScript Dates

Practice with Arrays

Formatting numbers with `toFixed()`

- `toFixed` will take care of rounding

```
let x = 3.141592653589;  
console.log(x.toFixed(0));          // returns 3  
console.log(x.toFixed(2));          // returns 3.14  
console.log(x.toFixed(4));          // returns 3.1416
```

```
let y = 8.899;  
console.log(y.toFixed(0));          // returns 9  
console.log(y.toFixed(2));          // returns 8.90
```

Converting Variables to Numbers

Number()	Returns a number, converted from its argument.
parseFloat()	Parses its argument and returns a floating point number
parseInt()	Parses its argument and returns an integer

Using parseInt() and parseFloat()

- Convert from String to number

```
console.log( parseFloat("11.11") );  
console.log( typeof parseFloat("11.11") );
```

```
console.log( parseInt("11.11") );  
console.log( typeof parseInt("11.11") );
```

11.11

number

11

number

Working with Built-In Objects and Arrays

Built-In Objects

Using String Properties and Methods

Using Number Methods



The JavaScript Math object and Random

Creating and displaying JavaScript Dates

Practice with Arrays

Math Object methods

- Various methods exist for mathematical operations

Math

[Back to Demos index](#)

Math.PI=3.141592653589793

Math.round(8.8)=9

Math.round(1.2)=1

Math.pow(2, 3)=8

Math.sqrt(49)=7

Math.abs(-1.4)=1.4

Math.ceil(3.4)=4

Math.floor(3.4)=3

Math.random

- Returns a random number between 0 (inclusive), and 1 (exclusive)

5 Random Values

Math.random()=0.5371599633604924

Math.random()=0.1142326889333447

Math.random()=0.7063359169244019

Math.random()=0.2442814283584045

Math.random()=0.40584058783037813

```
44     document.write('<br/>');
45     document.write('<br/>5 Values between 0 and 100');
46     for (let x = 0; x < 5; x++) {
47         document.write('<br/>');
48         document.write('Math.floor(Math.random() * 101)='
49                         + Math.floor(Math.random() * 101));
50     }
```

Getting a random integer value

- Can pass in the lower and upper bounds

```
54     document.write('<br/>call Random function :');
55     document.write(getRandomInteger(0,50));
56
57     function getRandomInteger(min, max) {
58         return Math.floor(Math.random() * (max - min)) + min;
59     }
```

Working with Built-In Objects and Arrays

Built-In Objects

Using String Properties and Methods

Using Number Methods

The JavaScript Math object and Random



Creating and displaying JavaScript Dates

Practice with Arrays

Creating a Date object with new Date()

- Calling new Date() gives the current data time information

```
<div id="displayNow"></div>

<script>
  let now = new Date();
  document.getElementById("displayNow").innerHTML = now;
</script>
```

Dates

[Back to Demos index](#)

Thu Mar 16 2017 16:51:56 GMT-0400 (Eastern Daylight Time)

Creating a Date object with new Date(milliseconds)

- Epoch is January 1, 1970 UTC

```
var milliSinceEpoch = new Date(90645645);
document.getElementById("displayNow").innerHTML = milliSinceEpoch;
```

Thu Jan 01 1970 20:10:45 GMT-0500 (Eastern Standard Time)

Creating a Date object with parameters

- Pass the year, (month-1), day, hour, minute, second, and millisecond
- JS uses 0 based index for months January = 0

```
let moonLanding = new Date(1969,6,20,20,18,0,0);
document.getElementById("displayNow").innerHTML = moonLanding;
```

Dates

[Back to Demos index](#)

Sun Jul 20 1969 20:18:00 GMT-0400 (Eastern Daylight Time)

Working with Built-In Objects and Arrays

Built-In Objects

Using String Properties and Methods

Using Number Methods

The JavaScript Math object and Random

Creating and displaying JavaScript Dates



Practice with Arrays

Creating a String from an Array

- Use `join()` with an optional delimiter to create a string from array elements

```
36 |   var letters2 = new Array("h","e","l","l","o");
37 |   document.write('<br/>Join array elements to a string');
38 |   document.write('<br/>');
39 |   document.write(letters2.join());
40 |   document.write('<br/>');
41 |   document.write(letters2.join('-'));
```

Join array elements to a string
h.e.l.l.o
h-e-l-l-o

Add and remove items to and from the end of an array

```
const bands = [];

bands.push('The Beatles');
bands.push('Aerosmith');
bands.push('The Temptations');

console.log(bands); // ["The Beatles", "Aerosmith", "The Temptations"]

const aBand = bands.pop()

console.log(aBand); // "The Temptations"
console.log(bands); // ["The Beatles", "Aerosmith"]
```

Sorting an Array

- The sort method will order of elements in the array, in place, by their string value's Unicode code point order.

```
const letters = ['z', 'q', 'e', 'a', 'q', 'j', i];
letters.sort(); // ["a", "e", "i", "j", "q", "q", "z"]
// As expected
```

```
var scores = [1, 10, 21, 2];
scores.sort(); // [1, 10, 2, 21] !!!
// What's going on here?
```

We will look at custom sorting in a later chapter

Return a portion of an array

- Use `slice()` to return a portion of an array

```
const values = ['a', 'b', 'c', 'd', 'e'];

values.slice() // ["a", "b", "c", "d", "e"]
values.slice(1) // ["b", "c", "d", "e"]
values.slice(1,3) // ["b", "c"]
values.slice(-2) // ["d", "e"]
values.slice(0,-3) // ["a", "b"]
```

Chapter Summary

In this chapter we have:

- Practiced with built-in objects and methods
- Used Number Methods
- Created random values
- Created and displayed JavaScript Dates
- Worked with Arrays

Chapter 5:

Browser Object Model

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Use more window object properties and methods
- Obtain info about the browser
- Get the current page address (URL)
- Redirect the browser to a new page
- Set Intervals and Timeouts

Browser Object Model



Using the window Object

Using the location Object

The Window Object

- Represents the browser's window
- Global JS objects, functions, and variables
 - Are members of the window object
 - And available for usage
- Example

```
window.document.getElementById("header");
```

is the same as:

```
document.getElementById("header");
```

Window.screen object

```
20 document.write("Screen Width: " + screen.width);
21 document.write('<br />');
22 document.write("Screen Height: " + screen.height);
23 document.write('<br />');
24 document.write("Available Screen Width: " + screen.availWidth);
25 document.write('<br />');
26 document.write("Available Screen Height: " + screen.availHeight);
27 document.write('<br />');
28 document.write("Color Depth(# of bits to display one color): " + screen.colorDepth);
```

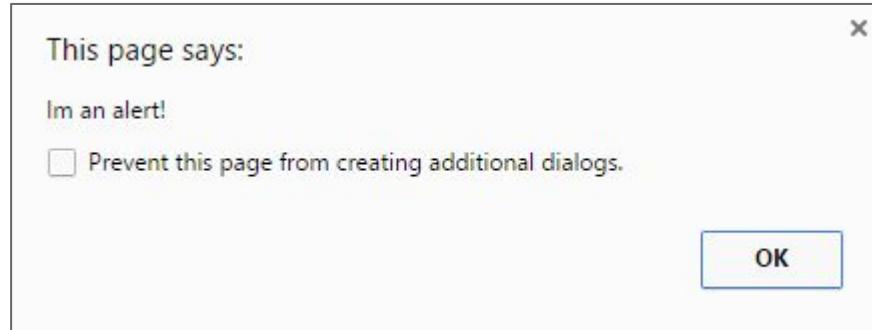
Window.navigator

```
25 let userAgentString = navigator.userAgent;
26 let platform = navigator.platform;
27 let language = navigator.language;
28 let online = navigator.onLine;
29
30 document.write('User Agent: ' + userAgentString);
31 document.write('<br />');
32 document.write('Platform: ' + platform);
33 document.write('<br />');
34 document.write('Language: ' + language);
35 document.write('<br />');
36 document.write('Is browser online?: ' + online);
```

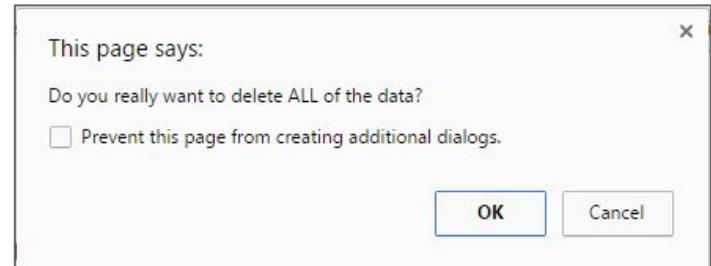
Using window.alert()

- Dialog boxes are modal windows

```
<input type="button" value="Click for an alert" onclick="alert('Im an alert!')">
```



Using window.confirm()



- Using confirm returns a true if OK is chosen, or false if Cancel

```
34     function confirmSomething() {  
35         let output = document.getElementById("confirmOutput");  
36         let outputText = '';  
37         if (window.confirm("Do you really want to delete ALL of the data?")) {  
38             outputText = 'You chose to delete it all!';  
39         }  
40         else {  
41             outputText = 'You hit cancel - whew!';  
42         }  
43         output.innerHTML = outputText;  
44     }  
45 }
```

Using `window.prompt()`

```
let result = window.prompt('Message to display', 'default');
```

- `result`
 - what the user puts into the prompt window. A string or null
- `message`
 - a string of text to display to the user
 - optional
- `default`
 - a string default value displayed in the text input field
 - It is an optional parameter

Example with prompt()

- Prompt can take a second argument for a default value
- Strings are read in, and must be converted if necessary

```
47  function promptSomething() {  
48      let output = document.getElementById("promptOutput");  
49      output.innerHTML='';  
50  
51      let name = prompt("What is your name?");  
52  
53      let experience =  
54          window.prompt('What level of experience do you have in JavaScript?', '1');  
55  
56      //convert string to numeric  
57      let moreExperience = +experience + 1;  
58  
59      output.innerHTML = name + " let's get you to a level " + moreExperience;  
60  }
```

Using Timers

- *setTimeout(functionName, time in milliseconds)*
- *clearTimeout(reference)*

```
<button onclick="aTimeoutReference = setTimeout(customTimeOut, 3000)">  
    Alert after 3 seconds</button>  
<button onclick="clearTimeout(aTimeoutReference)">ClearTimout Alert</button>
```

The setInterval() method

- Repeatedly calls a function or executes a code snippet with a fixed time delay between each call

```
78     let intervalID;
79     let counter = 1 ;
80     function intervalFunction() {
81         intervalID = window.setInterval(myCallback, 3000);
82     }
83     function cancelIntervalFunction() {
84         clearInterval(intervalID);
85     }
86     function myCallback() {
87         let intervalOutputDiv = document.getElementById("intervalOutput");
88         intervalOutputDiv.innerHTML += counter++;
89     }
```

Browser Object Model

Using the window Object



Using the location Object

Using window.location.href

- Can obtain the URL of the page

```
<div id="targetId"></div>
<script>
    document.getElementById("targetId").innerHTML =
        "Page URL is " + window.location.href;
</script>
```

Opening a Window

```
30     function loadMDN() {
31         window.location.assign("https://developer.mozilla.org/en-US/docs/Web/API/Window")
32     }
33 </script>
34
35 <input type="button" value="Reload Page with MDN: Window" onclick="loadMDN()">
```

Chapter Summary

In this chapter we have:

- Used more window object properties and methods
- Obtained info about the browser
- Got the current page address (URL)
- Redirected the browser to a new page
- Set Intervals and Timeouts

Chapter 6:

More work with the Document Object Model (DOM)

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Add and delete HTML elements
- Update the CSS with JS
- Work with Events

More work with the Document Object Model (DOM)



Registering and Reacting to Events

Adding and Removing DOM Nodes

Changing the style (CSS) of HTML elements

Examples of HTML events

- When a user clicks the mouse
- When a web page has loaded
- When an image has been loaded
- When the mouse moves over an element
- When an input field is changed
- When an HTML form is submitted
- When a user does a key press

Traditional (Inline) Model

- Can use HTML inline events

onchange	An HTML element has been changed
onclick	The user clicks an HTML element
onmouseover	The user moves the mouse over an HTML element
onmouseout	The user moves the mouse away from an HTML element
onkeydown	The user pushes a keyboard key
onload	The browser has finished loading the page

```
<button onclick="this.innerHTML=Date()">Click for the Time</button>
```

The onchange Event

- Often used with input fields

```
23    <label for="fname">Type in capital letters, then click away</label>
24    <input type="text" id="fname" onchange="lowerCase()">
25    <script>
26        function lowerCase() {
27            var x = document.getElementById("fname");
28            x.value = x.value.toLowerCase();
29        }
30    </script>
```

Mouse Events

- Each of the three mouse events can be handled separately
- The element and the event can be passed to a handler for processing

```
7          function process(element,event) {
8              element.value = 'Event ' + event.type ;
9          }
-- . . .

14         <form name="eventForm">
15             <input type="text" name="activity" onclick="process(this,event)"
16                 onmouseover="process(this,event)" onmouseout="process(this,event)"/>
17         </form>
```

Keyboard Events

- Keyboard events can be captured
 - Every key including control and shift generates an event
 - The event contains the key code and value

```
11      function processText(event) {  
12          document.eventForm.activity.value = 'Key ' + event.keyCode + " " + event.key ;  
13      }  
  
21      <input type="text" name="keyboard" onkeydown="processText(event)"/>
```

Multiple Click Handlers

- If there are multiple onclick handlers only the first gets executed

```
15         function processMultiple(element,message) {  
16             element.value += message ;  
17         }  
  
32         <input id="multiple" type="text" name="multiple" onclick="processMultiple(this,'1')"  
33             onclick="processMultiple(this,'2')"/>
```

Adding Event Listeners

- The addEventListener() method can separate JS from the markup
 - for better readability
 - add event listeners when you do not control the HTML markup
 - (such as with CMS systems)
- Can attach an event handler to the specified element
 - does not overwriting existing event handlers
 - can add **many** event handlers to one element, even same type

Adding Event Listeners

- The `addEventListener()` method makes it easier to control how the event reacts to bubbling
- When using the `addEventListener()` method, the JavaScript is separated from the HTML markup, for better readability and allows you to add event listeners even when you do not control the HTML markup.
- You can easily remove an event listener by using the `removeEventListener()` method.

Multiple Events

- Event handlers allow multiple event handlers on a single element

```
19      window.onload = function() {
20          const element = document.getElementById( "multiply" ) ;
21          element.addEventListener('click', function() { element.value += '1' } ) ;
22          element.addEventListener('click', function() { element.value += '2' } ) ;
23      }
34      <input id="multiply" type="text" name="multiply"/>
```

Event Bubbling

- Event bubbling is the normal operation
- When an event occurs it is passed down the DOM structure and the event handlers are called as the event propagates back up to the root
- The addEventHandler has an optional 3rd boolean parameter which defaults to false - bubble - the input handler called before the form handler

```
19         window.onload = function() {
20             const bubble = document.getElementById( "bubble" ) ;
21             document.getElementById( "multiply" ).addEventListener('click',
22                 function() { bubble.value += 'Text\n' }, false ) ;
23             document.getElementById( "eventForm" ).addEventListener('click',
24                 function() { bubble.value += 'Form\n' }, false ) ;
25         }
~~~
```

Event Capturing

- Event capturing is not often used
- When an event occurs it is passed down the DOM structure and the event handlers are called as the event goes down the DOM
- If the addEventHandler 3rd boolean parameter is set to true - capture - form handler is called before the input handler

```
19      window.onload = function() {
20          const bubble = document.getElementById( "bubble" ) ;
21          document.getElementById( "multiply" ).addEventListener('click',
22              function() { bubble.value += 'Text\n' }, true ) ;
23          document.getElementById( "eventForm" ).addEventListener('click',
24              function() { bubble.value += 'Form\n' }, true ) ;
25      }
```

Using `document.getElementsByTagName()`

- Returns an `HTMLCollection` of elements with the given tag name
- The complete document is searched, including the root node
- The returned `HTMLCollection` is live, meaning that it updates itself automatically to stay in sync with the DOM tree without having to call `document.getElementsByTagName()` again

More work with the Document Object Model (DOM)

Registering and Reacting to Events



Adding and Removing DOM Nodes

Changing the style (CSS) of HTML elements

Adding Nodes

- Nodes can be added at any point in the DOM
- A new node is created by tag type and added as a child to another element

```
27         function addOrder() {
28             const order = document.getElementById('order') ;
29             const starter = document.getElementById('starters') ;
30             const item = document.createElement('li') ;
31             item.innerHTML = starter.options[starter.selectedIndex].innerHTML ;
32             order.appendChild( item ) ;
33         }
34
35
36
37
38
39
40
41
42
43
44
45
46
47         <select id="starters" name="starters">
48             <option value="soup">Soup</option>
49             <option value="oysters">Oysters</option>
50             <option value="salmon">Smoked Salmon</option>
51         </select><br/>
52         <input type="button" onclick="addOrder();" value = "Order"/>
53     </form>
54     <ul id="order">
55     </ul>
-->
```

Removing Nodes

- A node can be removed by replacing outerHTML with an empty string

```
35      function remove() {  
36          const optional = document.getElementById('optional') ;  
37          optional.outerHTML = '' ;  
38      }  
  
64      <p id="optional">This is optional text</p>
```

More work with the Document Object Model (DOM)

Registering and Reacting to Events

Adding and Removing DOM Nodes



Changing the style (CSS) of HTML elements

Preventing Default Behavior

Styling with JavaScript

- Any of an element's styles can be changed
- Each element has a style attribute which allows individual styles to change

```
40      function colourMe() {
41          document.getElementById('colourme').style.color = document.changeCSS.colour.value ;
42      }
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70      <form name="changeCSS">
71          <input type="radio" name="colour" value="red" checked />Red<br/>
72          <input type="radio" name="colour" value="green" />Green<br/>
73          <input type="radio" name="colour" value="blue" />Blue<br/>
74          <input type="radio" name="colour" value="custom" />Custom<br/>
75          <input type="button" onclick="colourMe();" value = "Colour"/>
76      </form>
77      <p id="colourme">This is colourful text</p>
```

Changing Classes

- CSS classes can be set, added or removed from an element
 - Set, append or clear the className attribute

```
44         function changeClass() {
45             const element = document.getElementById('styleme') ;
46             const action = document.changeClass.colour.value ;
47             if ( action == "Set" ) {
48                 element.className = "setSize"
49             } else if ( action == "Add" ) {
50                 element.className += " setColour" ;
51             } else {
52                 element.className = "" ;
53             }
54 }
```

Chapter Summary

In this chapter we have:

- Added and deleted HTML elements
- Updated the CSS with JS
- Worked with Events

Chapter 7:

Using JavaScript with Forms

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Work with data from form inputs
- Stop form submissions
- Validate Inputs
- Provide feedback to users

Form Validation with JavaScript



Work with data from form inputs

Stopping Form Submission

Validating Form Fields

Provide feedback to users

Accessing Forms

- Forms are part of the DOM
- They can be accessed in a number of ways
 - If the form has an id can use document.getElementById
 - If the form is named it can be accessed from the forms array
 - If the form is named it can be accessed by name

```
<form id="form1" action="done.html"
      name="validateform"
      onsubmit="return validate();">
```

```
document.getElementById( 'form1' ) ;
document.forms['validateform'] ;
document.validateform ;
```

Accessing Form Input Controls

- Form input controls can be accessed from the form object
- It can be accessed by indexing the form with its name
- It can be accessed directly by name
- Inputs have a value attribute which can be used to read and change the value

```
alert( document.forms['validateform']['name'].value ) ;  
alert( document.validateform.name.value ) ;
```

Disabling controls

- Form controls can be disabled in HTML
- They can also be enabled and disabled in JavaScript

```
74      <input type="checkbox" name="cream" value="Cream" disabled>Cream<br/>
```

```
8          document.radioform.cream.disabled = true ;
9          document.radioform.cream.disabled = true ;
```

Getting values from different control types

- Most controls have a value attribute
- The value is often assigned in the HTML

```
67 <form id="form2" action="done.html" name="radioform" onsubmit="return validateOption();">
68   <select id="starters" name="starters">
69     <option value="soup">Soup</option>
70     <option value="oysters">Oysters</option>
71     <option value="salmon">Smoked Salmon</option>
72   </select><br/>
73   <input type="radio" name="choose" value="Cake" onclick="chooseOption(this); checked />Cake<br/>
74   <input type="radio" name="choose" value="Scone" onclick="chooseOption(this); />Scone
75   <input type="checkbox" name="cream" value="Cream" disabled/>Cream<br/>
76   <input type="radio" name="choose" value="Other" onclick="chooseOption(this); />Other
77   <input type="text" id="dessert" name="dessert" style="visibility:hidden"/><br/>
78   <textarea name="order" rows="3" cols="20"></textarea><br/>
79   <input type="button" onclick="showOrder();" value = "Order"/>
```

Text, TextArea, and Select

- The `text` and `textarea` controls can have their contents changed by writing to the value
- The `select` control's option text can be extracted from the `innerHTML` of the selected index

```
43     function showOrder() {  
44         const order = document.radioform.order ;  
45         const starter = document.getElementById('starters') ;  
46         order.value = starter.options[starter.selectedIndex].innerHTML + "\n"  
47             + document.radioform.choose.value ;  
48     }
```

Control Visibility

- Controls can be made visible or invisible based on the state of other controls
- There is a style called visibility which can be set to hidden or visible in both HTML and JavaScript

77

```
<input type="text" id="dessert" name="dessert" style="visibility:hidden"/><br/>
```

```
document.radioform.desert.style.visibility = element.value == "Other" ? "visible" : "hidden" ;
```

Example of making 'Other' input field appear

- A field can be made to appear or disappear depending on another control
 - Create radio buttons with the first one selected
 - Radio buttons call a function with their this as a parameter using onclick
 - The text box is initially styled to be invisible
 - When a radio button is clicked the function makes the text box visible only if "Other" is selected
 - If "Other" is selected validation is required for the text box content

<input type="radio"/> option 1 (selected)
<input type="radio"/> option 2
<input checked="" type="radio"/> Other:

Form Validation with JavaScript

Work with data from form inputs



Stopping Form Submission

Validating Form Fields

Provide feedback to users

Form Submission

- Users often make mistakes when filling in web page forms
- Need to give the user fast feedback of any errors
- JavaScript can validate form input values
- The form submit can be stopped to avoid unnecessary network traffic and server side processing

Form onsubmit Option

- The HTML form tag has an onsubmit attribute
- If the onsubmit attribute has a false value the form submit is stopped
- The onsubmit usually calls a validation function

```
64      <form id="form1" action="done.html" name="validateform" onsubmit="return validate();">
65          Name: <input type="text" name="name" /><span id="name_error" class="error"></span><br/>
66          Zip Code: <input type="text" name="zip" /><span id="zip_error" class="error"></span><br/>
67          <input type="submit" name="submit" value="Enter" />
68      </form>
```

Validation Function

- The validation function returns true or false which decides if the submit occurs

```
9      function validate() {  
10         alert( "Stopping form submission" ) ;  
11         return false ;  
12     }
```

More than returning false...

- Attach an event listener to the form using `.addEventListener()` and then call the `.preventDefault()` method on event:

```
window.onload = function() {
    document.getElementById( "form3" ).addEventListener( "submit", function( event ) {
        alert( "Stopping commit" ) ;
        event.preventDefault() ;
    });
};
```

Form Validation with JavaScript

Work with data from form inputs

Stopping Form Submission



Validating Form Fields

Provide feedback to users

Form Validation

- Form validation can take a number of forms
 - Have all of the required fields been filled in?
 - Have fields been given the correct date format?
 - Do numeric fields contain invalid numbers?

Validation Errors - Missing Fields

- What to do if a required fields is empty?
 - Report the error to the user
 - Put the focus on the missing control
 - Stop the form from being submitted

```
7      function validate() {
8          var result = true ;
9          if ( document.validateform.name.value == "" ) {
10              alert( "Please enter your name" ) ;
11              document.validateform.name.focus() ;
12              result = false ;
13          }
14          return result ;
15      }
```

Validation Errors - Incorrect Fields

- What to do if a required fields is incorrect?
 - Report the error to the user
 - Put the focus on the missing control
 - Clear the control
 - Stop the form from being submitted

```
15 |     if ( document.validateform.zip.value == "" ||  
16 |             isNaN( document.validateform.zip.value ) ||  
17 |             document.validateform.zip.value.length != 5 ) {  
18 |                 alert( "Please enter your 5 digit zip code" ) ;  
19 |                 document.validateform.zip.value = "" ;  
20 |                 document.validateform.zip.focus() ;  
21 |                 result = false ;  
22 }
```

Other Validation

- Validation is often performed when a form is submitted
 - Call a function using the `onsubmit` attribute
 - The function returns a boolean, `false` stops the submit
- Validation can also be performed when a control is changed
 - Call a function using the `onclick` attribute
 - The function can be passed the control object using `this`

```
<input type="button" onclick="showOrder();" value = "Order"/>
```

Other Validation

- Validation can also be performed when a control is changed
 - Call a function using the `onclick` attribute
 - The function can be passed the control object using `this`
 - This is useful for radio buttons

```
<input type="radio" name="choose" value="Cake" onclick="chooseOption(this); checked />Cake<br/>
<input type="radio" name="choose" value="Scone" onclick="chooseOption(this); />Scone
```

```
function chooseOption(element) {
    document.radioform.cream.disabled = element.value != "Scone" ;
    document.radioform.desert.style.visibility = element.value == "Other" ? "visible" : "hidden" ;
}
```

Form Validation with JavaScript

Work with data from form inputs

Stopping Form Submission

Validating Form Fields



Provide feedback to users

How to Feedback Validation Errors

- Display a meaningful error message
- Next to the incorrect control or an alert?
- Move the focus to the invalid control
- Clear any incorrect text in the control?

Displaying the Error Message

- The error message can be displayed in a suitably styled element
- Create a stylesheet called `style.css` containing the `style` class for the error
- Link the stylesheet into the HTML
- Add an empty control for the error message

```
1 .error {  
2     color: #FF0000;  
3 }
```

5

```
<link rel="stylesheet" type="text/css" href="style.css"/>
```

Zip Code: `<input type="text" name="zip" />
`

Display the Error Message on Validate

- Update the styled error element to display the message

```
17     document.getElementById("zip_error").innerHTML = "" ;
18     if ( document.validateform.zip.value == "" ||
19         isNaN( document.validateform.zip.value ) ||
20         document.validateform.zip.value.length != 5 ) {
21         document.getElementById("zip_error").innerHTML = "Please enter your 5 digit zip code" ;
22         document.validateform.zip.value = "" ;
23         document.validateform.zip.focus() ;
24         result = false ;
25     }
```

Chapter Summary

In this chapter we have:

- Work with data from form inputs
- Stop form submissions
- Validate Inputs
- Provide feedback to users

Chapter 08:

Object Oriented JavaScript

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Become more familiar with object-oriented JavaScript
- Create objects in multiple ways
- Look at prototypical inheritance

Object Oriented JavaScript



Reviewing Object Creation

Constructor Functions

Basics of prototypes

Recall using Object Literals or using new Object

- Here an Object is created as an Object Literal - using curly braces

```
var band1 = {  
    "name" : "Pink Floyd",  
    "country" : "England",  
    "yearFormed" : 1965,  
    "genres" : ["Progressive rock", "psychedelic rock", "art rock"]  
}
```

- This is the equivalent of:

```
var band2 = new Object();  
band2.name = "Pink Floyd";  
band2.country = "England";  
band2.yearFormed = 1965;  
band2.genres = ["Progressive rock", "psychedelic rock", "art rock"];
```

Properties can be referenced with dot or brackets

- Properties can be referenced using dot notation or bracket notation

```
//access properties using dot notation or □  
console.log("country 1 = " + band1.country);  
console.log("country 2 = " + band2["country"]);
```

Methods are functions in objects

- A method is simply a key with a function as a value
- A method is called by the key with parentheses

```
15      const Person = {  
16          "firstName": "Carole",  
17          "lastName": "King",  
18          "fullName": function() {  
19              return this.firstName + ' ' + this.lastName ;  
20          }  
21      }
```

```
32           console.log( "Person " + Person.fullName() ) ;
```

Can use a for loop to get object properties

- A for loop can be used to iterate over an object's properties

```
6      const band1 = {  
7          "name": "Pink Floyd",  
8          "country": "England",  
9          "yearFormed": 1965,  
10         "genres": ["Progresssive Rock", "Psychedelic Rock", "Art Rock"]  
11     }  
12  
13     function showBands() {  
14         for ( let property in band1 ) {  
15             document.bands.bandData.value += band1[property] + "\n" ;  
16         }  
17     }
```

Object Oriented JavaScript

Ways to Create Objects



Constructor Functions

The context of this

Basics of prototypes

Creating Objects with Constructor functions

- Objects can be created from constructor functions
- Objects are instantiated using the `new` keyword
 - If not called with `new` then `this` is the window and not the object

```
19      function Student( firstName, lastName ) {  
20          this.firstName = firstName ;  
21          this.lastName = lastName ;  
22      }  
  
25      const student = new Student( "Peter", "Jones" ) ;
```

To prevent constructor being called without ‘new’

- The constructor can check the type of `this` to ensure it is called correctly

```
19         function Student( firstName, lastName ) {
20             if ( ! ( this instanceof Student ) ) {
21                 return new Student( firstName, lastName ) ;
22             }
23             this.firstName = firstName ;
24             this.lastName = lastName ;
25             this.fullName = function() {
26                 return this.firstName + ' ' + this.lastName ;
27             }
28         }
```

Functions in constructor functions

- The constructor function can include functions

```
19         function Student( firstName, lastName ) {
20             this.firstName = firstName ;
21             this.lastName = lastName ;
22             this.fullName = function() {
23                 return this.firstName + ' ' + this.lastName ;
24             }
25         }
```

32

```
let fullName = student.fullName() ;
```

Object Oriented JavaScript

Ways to Create Objects

Constructor Functions



The context of this

Basics of prototypes

Binding of this

- Binding of ‘this’ happens when code is called, not defined
- The context is based on how you invoke it

```
18  function doThing() {  
19      console.log("Name:", this.name);  
20  }  
21  
22  let student1 = {name: "Casey", doThing: doThing};  
23  let student2 = {name: "Anubhav", doThing: doThing};  
24  
25  student1.doThing(); // Name: Casey  
26  student2.doThing(); // Name: Anubhav  
27  doThing(); // Name: undefined
```

Constructor Functions in JS are not like Java

```
33  function Person() {
34      // The Person() constructor defines `this`
35      // as an instance of itself.
36      this.age = 0;
37
38      this.getOlder = function() {
39          // `this` is the global object,
40          // different from the `this`
41          // defined by the Person() constructor
42          this.age++;
43          console.log(age);
44      };
45  }
46
47 var student = new Person();
48 student.getOlder();
```

✖ Uncaught ReferenceError: age is not defined
at Person.getOlder ([this.html:43](#))
at [this.html:48](#)

Object Oriented JavaScript

Ways to Create Objects

Constructor Functions

The context of this



Basics of prototypes

The assignment operator calls Object.create()

- Object.create() takes a prototype parameter
- The default is Object.prototype
- The two examples are equivalent

```
47         const student = { firstName: "Peter", lastName: "Jones" } ;
```

```
54         const student = Object.create( Object.prototype ) ;
55         student.firstName = "Peter" ;
56         student.lastName = "Jones" ;
```

Object.prototype provides functions

key	value
constructor	Object
toString	function() { [native code] }
toLocaleString	function() { [native code] }
valueOf	function() { [native code] }
hasOwnProperty	function() { [native code] }
isPrototypeOf	function() { [native code] }
propertyIsEnumerable	function() { [native code] }

key	value
firstName	“Hahn”
lastName	“Bui”

Creating and using custom prototype

- Put properties that don't change for new instances into a prototype object
- New instances can then be constructed from the prototype

```
63      const pupil = {} ;
64      pupil.yearEnrolled = 2017 ;
65      const student = Object.create( pupil ) ;
66      student.firstName = "Peter" ;
67      student.lastName = "Jones" ;
-- -- -- -- --
```

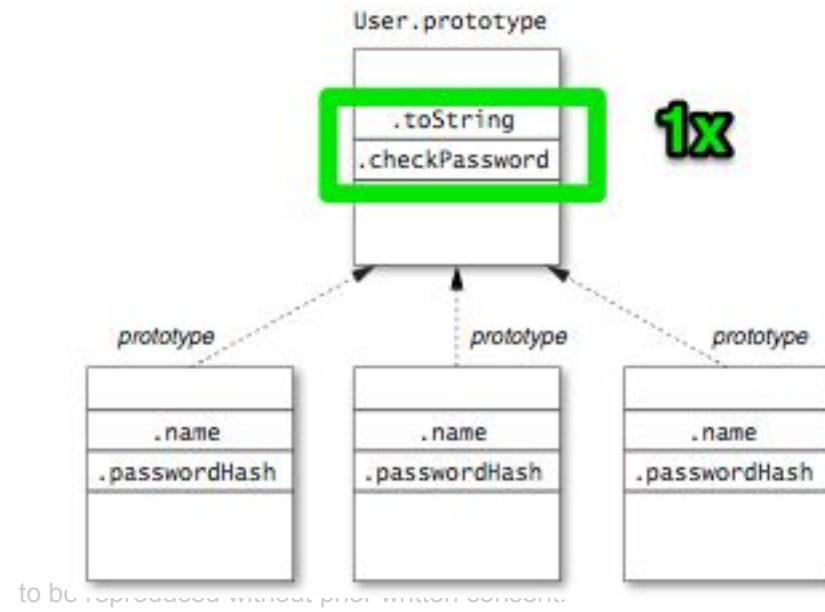
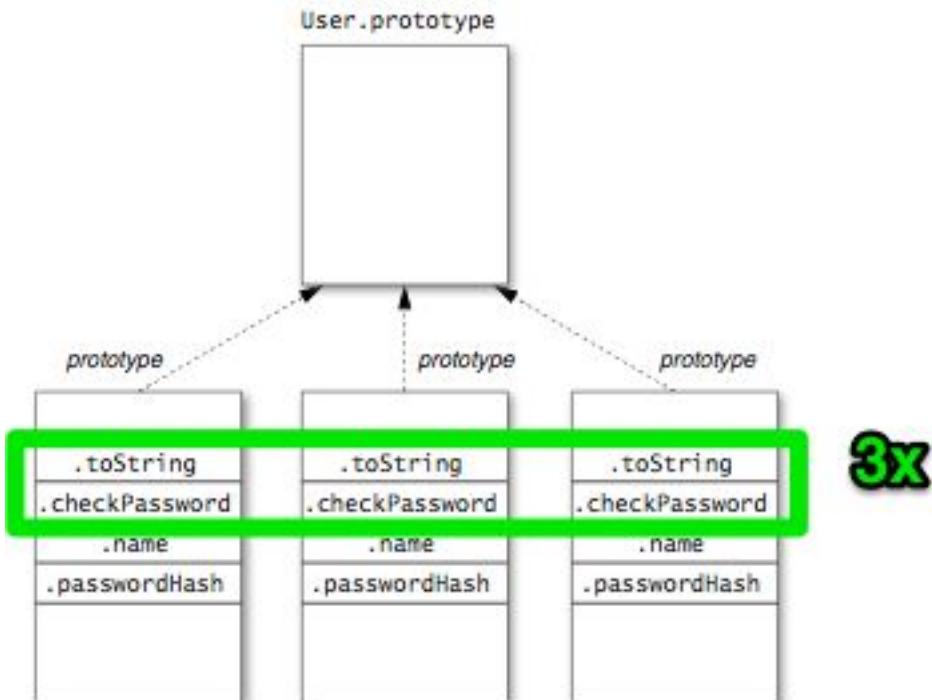
Can add functions to prototypes

- Functions can be added to an object's prototype
- The `toString()` method can be replace
 - It is called when the object is accessed in string context

```
74          Student.prototype.toString = function() {  
75              return this.firstName + ' ' + this.lastName ;  
76      }  
77      const student = new Student( "Peter", "Jones" ) ;  
78      document.students.studentData.value += student + "\n" ;  
79  }
```

Difference in memory storage

difference in memory storage between instance and prototype functions?



Chapter Summary

In this chapter we have:

- Became more familiar with object-oriented JavaScript
- Created objects in multiple ways
- Looked at prototypical inheritance

Chapter 09: **More Functions** **and Code Organization**

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Review more advanced features of functions in JS
- Use callbacks
- Discuss closures
- Work with IIFE
- Practice with arrow functions

More Functions and Code Organization



Using callbacks and anonymous functions

Arrow functions

Closures explained

Immediately Invoked Function expressions (IIFE)

Modular JavaScript

Higher-order functions

- Functions that can take other functions as arguments are called higher-order functions
- Allow for abstractions and more focused code writing
- Start with a simple example and build on it with functions

```
6      const starters = ['Soup', 'Fois Gras', 'Smoked Salmon', 'Oysters'] ;
7
8      for ( const dish of starters ) {
9          console.log( dish ) ;
10     }
```

Wrap the loop in a function

- Create a function which is passed the array as a parameter
 - This decouples the function from the array

```
12          function logArray( array ) {  
13              for ( const item of array ) {  
14                  console.log( item ) ;  
15              }  
16          }  
17          logArray( starters ) ;
```

Pass a function into a function

- Now pass the processing function as a parameter

```
19      function processArray( array, action ) {
20          for ( const item of array ) {
21              action( item ) ;
22          }
23      }
24      processArray( starters, console.log ) ;
```

Rewrite Using Anonymous Function

- A function parameter can be replaced by an anonymous function

```
26      processArray( starters, function( entry ) {  
27          console.log( entry ) ;  
28      } ) ;
```

Custom sorting of Arrays

- To override JavaScript's default sort order you can use a custom compare function.

```
numbers.sort(function(a, b) {  
  if (a < b) {  
    // If the first element is less than the second return a negative number  
    return -1;  
  } else if (a > b) {  
    // If the first element is greater than the second return a positive number  
    return 1;  
  } else {  
    // If they are equal, return 0  
    return 0  
  }  
});  
// [1, 2, 10, 21]
```

```
numbers.sort(function(a, b) {  
  // In this case a - b also yields the same result  
  return a - b;  
});
```

Convert an array to another

- The `map()` function uses the supplied *callback* function for each item in the array to convert it to another array.

```
const values = [1,2,3,4,5];

const items = values.map(function(value) {
  return "Item: " + value;
});

console.log(items);
// Produces ["Item: 1", "Item: 2", "Item: 3", "Item: 4", "Item: 5"]
```

Find an element in an array

- The `find()` function returns the first element in the array that where the predicate function returns `true`.

```
const haystack = ['Apple', 'Banana', 'Cherry'];

const needle = haystack.find(function(currentValue) {
    return currentValue.startsWith('B');
});

console.log('Needle is: ' + needle);
// Produces: "Needle is: Banana"
```

Filter an array

- Select all of the items in an array that match a predicate function with `filter()`:

```
const numbers = [1,2,3,4,5,6];

const evens = numbers.filter(function(currentValue) {
    return currentValue % 2 == 0;
});

console.log('Evens are: ' + evens.join(', '));
// Produces: "Evens are: 2, 4, 6"
```

Perform an action for each element in an array

- `forEach()` executes the supplied function on each element of an array:

```
[1,2,3,4,5].forEach(function(currentValue) {  
  console.log(`The current value is ${currentValue}`);  
});  
  
// Produces:  
// "The current value is 1"  
// "The current value is 2"  
// "The current value is 3"  
// "The current value is 4"  
// "The current value is 5"
```

Reduce an array to a single value

- Applies a function to each element of an array, and an accumulator value, returning a single value

```
const values = [10, 40, 50];
let sum = 0;

sum = values.reduce(function(value, accum) {
  return accum + value;
}, sum);

console.log(`Sum is ${sum}`);
// Produces: "Sum is 100"
```

More Functions and Code Organization

Using callbacks and anonymous functions



Arrow functions

Closures explained

Immediately Invoked Function expressions (IIFE)

Modular JavaScript

Arrow Functions

- An arrow function is a shortened version of an anonymous function
 - If there is a single parameter parentheses are not required
 - If there is a single body statement braces are not required

```
30      processArray( starters, (entry) => { console.log( entry ) } ) ;
31      processArray( starters, entry => { console.log( entry ) } ) ;
32      processArray( starters, entry => console.log( entry ) ) ;
```

Using Arrow Functions

- Some of the array examples can be rewritten using arrow functions

```
21      const values = [1, 2, 3, 4, 5];
22
23      const items = values.map((value) => {
24          return "Item: " + value;
25      );
26
27      console.log(items);
```

```
▶ ["Item: 1", "Item: 2", "Item: 3", "Item: 4", "Item: 5"]
```

More Arrow Functions

- Using arrow functions as a filter

```
39      const moreNumbers = [1, 2, 3, 4, 5, 6];
40
41      const evens = moreNumbers.filter((currentValue) => {
42          return currentValue % 2 == 0;
43      );
44
45      console.log('Evens are: ' + evens);
```

Evens are: 2,4,6

More Functions and Code Organization

Using callbacks and anonymous functions

Arrow functions

➤ **Closures explained**

Immediately Invoked Function expressions (IIFE)

Modular JavaScript

Closures are a powerful feature in JavaScript

- A closure is a function which is defined inside another function
 - inner function has access to outer function's variables
 - even after the outer function exits
 - Allows variables and functions to be in different scopes
 - lexical scoping
- PROs
 - Data can be associated with a function
 - It is more of an object oriented approach
 - They allow private methods to be created

Functions inside function - Lexical Scoping

- A function can be defined inside a function
 - Outer function's variables and inner function are not visible outside
 - Inner function can access the outer function's variables

```
34         function init() {  
35             const main = "Chicken Kiev" ;  
36             function showMain() {  
37                 console.log( main ) ;  
38             }  
39             showMain() ;  
40         }  
41         init() ;
```

Closure

- A closure a combination of a function and its lexical environment
 - The function returns the inner function
 - The outer function's variables are hidden from outside but can be accessed by the inner function

```
43         function makeMain() {  
44             const main = "Chicken Kiev" ;  
45             function showMain() {  
46                 console.log( main ) ;  
47             }  
48             return showMain ;  
49         }  
50         const myMain = makeMain() ;  
51         myMain() ;
```

Anonymous Closure Function

- A closure can return an anonymous function
 - The inner function can access the outer function's parameters

```
53         function makeMultiplier( x ) {
54             return function( y ) {
55                 return x * y ;
56             }
57         }
58         const multiplyBy5 = makeMultiplier( 5 ) ;
59         console.log( multiplyBy5( 6 ) ) ; // 30
60         const multiplyBy7 = makeMultiplier( 7 ) ;
61         console.log( multiplyBy7( 3 ) ) ; // 21
```

More Functions and Code Organization

Using callbacks and anonymous functions

Arrow functions

Closures explained



Immediately Invoked Function expressions (IIFE)

Modular JavaScript

Immediately Invoked Function Expression (IIFE)

```
22  var agingService = function () {  
23      let age = 0;  
24  
25      this.showAge = function () {  
26          age++;  
27          console.log('age is ' + age);  
28      };  
29  
30      return this;  
31  }();  
32  
33  agingService.showAge();  
34  agingService.showAge();
```

- Function is converted into an object right after it is parsed
- Parenthesis causes immediate execution
- Common in older JavaScript patterns

age is 1
age is 2

Various Patterns were developed to organize code

- Revealing Module Pattern
 - Uses an IIFE
 - Exploses an object
- Used a lot in jQuery plugins to organize code and avoid conflicts
- Newer approach in modern JavaScript is to use modules

```
22  var messageUtility = function () {  
23      var message = 'This is the message!';  
24      function printMessage() {  
25          console.log(message);  
26      }  
27      function updateMessage(newMessage) {  
28          message = newMessage;  
29      }  
30      //returns object representing module  
31      //Revealing Module Pattern  
32      return {  
33          showMessage: printMessage,  
34          updateMessage: updateMessage  
35      }  
36  }(); //← IIFE
```

More Functions and Code Organization

Using callbacks and anonymous functions

Arrow functions

Closures explained

Immediately Invoked Function expressions (IIFE)



Modular JavaScript

Why Modules?

- Simplify dependency management
- Code Organization: divide up functionality of app & provide encapsulation
- Code reusability
- Decoupling
- Code Extensibility

Modules

- Modules organize code and prevent polluting the global namespace
- Modules are executed within their own scope, not in the global scope
 - variables, functions, classes, declared in module not visible
 - unless they are explicitly exported
- To use an exported item in a different module, it has to be imported

Example Module Export and Import

```
3 module.exports = function arrayToText(array) {  
4     return array.join(", ");  
5 }
```

array-to-text.js

```
5 const arrayToText = require("./array-to-text");  
6  
7 document.addEventListener("DOMContentLoaded", function() {  
8  
9     var text = arrayToText(combinedArray);  
10  
11     document.getElementById("text").innerHTML = text;  
12 });
```

example.js

Before ES6 (ES2015): two standards

- **CommonJS Modules:**
 - What is used in Node.js
 - Compact syntax
 - Designed for synchronous loading and servers
- **Asynchronous Module Definition (AMD):**
 - What is used in RequireJS
 - More complicated syntax
 - enabling AMD to work without eval() or a compilation step
 - Designed for asynchronous loading and browsers

ECMAScript 6 modules

- ES6 goal was a format users of CommonJS and AMD were happy with
- Like CommonJS: compact syntax, a preference for single exports and support for cyclic dependencies
- Like AMD: support for asynchronous loading and configurable module loading
- ES6 modules go beyond CommonJS and AMD

Module loaders

- Loaders interpret and load a module written in a certain module format
- A module loader runs at runtime:
 - you load the module loader in the browser
 - you tell the module loader which main app file to load
 - the module loader downloads and interprets the main app file
 - the module loader downloads files as needed
- Popular loaders:
 - **RequireJS**: for modules in AMD format
 - **SystemJS**: for modules in AMD, CommonJS, UMD or System.register

Module Loaders

- At runtime the module loader is responsible for locating and executing all dependencies of a module before executing it
- Well-known modules loaders:
 - CommonJS module loader for Node.js
 - require.js for Web applications

Module bundlers replace module loaders

- A module bundler runs at build time
 - generates a bundle file at build time - ex bundle.js
- Load the bundle in the browser
 - No module loader is needed in the browser
 - All code is included in the bundle.
- Examples of popular module bundlers are:
 - **Browserify**: bundler for CommonJS modules
 - **Webpack**: bundler for AMD, CommonJS, ES6 modules

Client-side Node with Browserify

- Allows CommonJS modules to be compiled for use in the browser
- <http://browserify.org/>



Chapter Summary

In this chapter we have:

- Reviewed more advanced features of functions in JS
- Used callbacks
- Discussed closures
- Worked with IIFE
- Practiced with arrow functions

Chapter 10:

JSON

Introduction to Modern JavaScript



Chapter Objectives

In this chapter we will:

- Review JSON syntax
- Compare JSON to XML
- Work with JSON parse functions
- Stringify JSON objects to strings

JavaScript Object Notation (JSON)



What is JSON Syntax?

JSON vs XML

JSON Processing & Parsers

What is JSON? (JavaScript Object Notation)

- JSON data is written as name/value pairs
- A name/value pair consists of:
 - a field name (in **double** quotes), followed by a colon, followed by a value
- Example

```
"city": "Boston"
```

JSON names **require** double quotes

JavaScript names do not

Sources for JSON

- A common use of JSON is to read data from a web server, and display the data in a web page.
- Text files
- In web files: Be careful editing! Line breaks such as invisible \n is bad!

```
var bandJSONStringData = '{"name" : "Pink Floyd", ' +  
    '"city" : "London", ' +  
    '"country" : "England", ' +  
    '"yearFormed" : "1965"}';  
  
var bandObj = JSON.parse(bandJSONStringData);
```

- Server calls, like AJAX

JSON Values Can Be:

- A number (integer or floating point)
- A string (in double quotes)
- A Boolean (true or false)
- An array (in square brackets)
- An object (in curly braces)
- null

```
{  
  "bands":{  
    "band": [  
      {  
        "name": "Pink Floyd",  
        "city": "London",  
        "country": "England",  
        "yearFormed": "1965",  
        "genres": [ "Progressive rock", "psychedelic rock", "art rock"]  
      },  
      {  
        "name": "Morphine",  
        "country": "USA",  
        "city": "Boston",  
        "genres": [ "Alternative rock", "experimental rock", "jazz rock"]  
      },  
      {  
        "name": "Vitamin String Quartet",  
        "country": "USA",  
        "city": "Los Angeles",  
        "genres": [ "Rock", "Pop", "Instrumental"]  
      }  
    ]  
  }  
}
```

Tools

- JSON formatters, validators and converters
 - <https://jsonformatter.curiousconcept.com/>
 - <http://www.utilities-online.info/xmltojson>
 - <http://www.freeformatter.com/json-formatter.html>
- IDE Plugins:
 - Eclipse - [JSON Tools](#)

DO NOW: Explore Tools and JSON Data

10 min

- Visit the URL for itunes <https://itunes.apple.com/lookup?id=909253>
 - View the data returned
- Open the URL: <http://www.utilities-online.info/xmltojson>
- Copy and Paste the JSON data, transform to XML.
- Do you prefer the XML or the JSON?
- Think About it...
 - How could you use this data to create an information page for music?
 - What pieces of data would you use?

JavaScript Object Notation (JSON)

What is JSON Syntax?



JSON vs XML

JSON Processing & Parsers

XML versus JSON

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <bands>
3    <band>
4      <name>Pink Floyd</name>
5      <city>London</city>
6      <country>England</country>
7      <yearFormed>1965</yearFormed>
8      <genres>Progressive rock</genres>
9      <genres>psychedelic rock</genres>
10     <genres>art rock</genres>
11   </band>
12   <band>
13     <name>Morphine</name>
14     <country>USA</country>
15     <city>Boston</city>
16     <genres>Alternative rock</genres>
17     <genres>experimental rock</genres>
18     <genres>jazz rock</genres>
19   </band>
20   <band>
21     <name>Vitamin String Quartet</name>
22     <country>USA</country>
23     <city>Los Angeles</city>
24     <genres>Rock</genres>
25     <genres>Pop</genres>
26     <genres>Instrumental</genres>
27   </band>
28 </bands>
```

```
1 { "bands": [
2   {
3     "band": [
4       {
5         "name": "Pink Floyd",
6         "city": "London",
7         "country": "England",
8         "yearFormed": "1965",
9         "genres": [
10           "Progressive rock", "psychedelic rock", "art rock"
11         ]
12       },
13       {
14         "name": "Morphine",
15         "country": "USA",
16         "city": "Boston",
17         "genres": [
18           "Alternative rock", "experimental rock", "jazz rock"
19         ]
20       },
21       {
22         "name": "Vitamin String Quartet",
23         "country": "USA",
24         "city": "Los Angeles",
25         "genres": [
26           "Rock", "Pop", "Instrumental"
27         ]
28       }
29     ]
30   }
31 ]}
```

Similarities/Differences between JSON & XML

- Both are called "self describing" (human readable)
 - as long as tags and names are descriptive
- Both have a hierarchy
 - JSON can use arrays and translates directly into Javascript objects
- Both can be parsed and used by lots of programming languages
 - JSON support is built in to JavaScript & Python

More discussion at:

<http://www.json.org/xml.html>

JavaScript Object Notation (JSON)

What is JSON Syntax?

JSON vs XML



JSON Processing & Parsers

JSON Parsers

- JSON is primarily a string of text
- Needs to be converted to an object to make it useful
- Need to utilize a JSON parser, built in to JavaScript

```
var bandJSONStringData = '{"name" : "Pink Floyd", ' +  
    '"city" : "London", ' +  
    '"country" : "England", ' +  
    '"yearFormed" : "1965"}';  
  
var bandObj = JSON.parse(bandJSONStringData);
```

The JSON Object has two methods

- `JSON.parse(aString[, reviver])`
 - turns JSON strings into JavaScript Objects
- `JSON.stringify(objectJSON, [, replacer[, space]])`
 - turns a JS object into a JSON string

Web Browsers Support

Firefox 3.5
Internet Explorer 8
Chrome
Opera 10
Safari 4

A simple parse using JSON.parse(aString)

- This is a simple parse.

```
var data = '{"name": "John", "email": "", "phone": "555-1212"}';
var o1 = JSON.parse(data);
console.log(o1);
```

```
▼ Object ⓘ
  email: ""
  name: "John"
  phone: "555-1212"
```

A parse using `JSON.parse(aString[, reviver])`

- Reviver is optional, and can be used to translate values into something meaningful

```
var o = JSON.parse(data, function(k, v) {  
    if (k === '') { return v; } // if topmost value, return it,  
    console.log(k);  
    if (v == '') {  
        return "unknown";  
    }  
    else return v;  
});  
console.log(o);
```

▼ Object ⓘ
email: "unknown"
name: "John"
phone: "555-1212"

Processing Data from a Server

- We will be making AJAX calls in class to different servers
- When receiving data, we need to parse the results

Chapter Summary

In this chapter we have:

- Reviewed JSON syntax
- Compared JSON to XML
- Worked with JSON parse functions
- Stringified JSON objects to strings

Chapter 11:

Asynchronous Programming

with JavaScript

Introduction to Modern JavaScript

Chapter Objectives

In this chapter we will:

- Make asynchronous calls to a server using JS
- Configure AJAX requests
- Handle errors
- Introduce Promises

Asynchronous Programming with JavaScript



Overview of AJAX

The XMLHttpRequest Object

Configuring an AJAX Request

Handling Errors on readystatechange

Introduction to Promises

Overview of AJAX

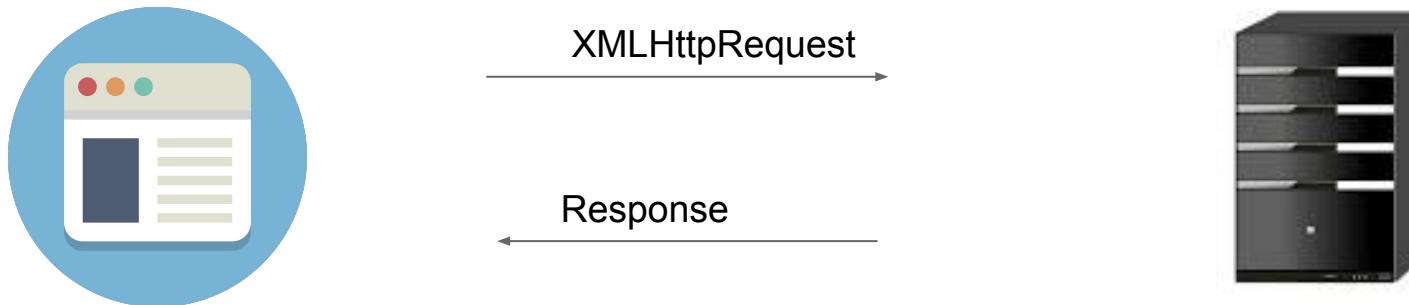
- Asynchronous JavaScript And XML
 - It's not separate code - its JavaScript!
- AJAX is a combination of:
 - A browser built-in XMLHttpRequest object
 - (to request data from a web server)
 - JavaScript and HTML DOM
 - (to display or use the data)

With AJAX, you can:

- Update a portion of a web page without full reload
- Continue to request fresh data after a page has loaded
- Send information in the background to be saved on the server

How AJAX works

- The client JavaScript makes an XMLHttpRequest and the server sends an HTTP Response
 - The response data is used to update part of the web page rather than replacing the whole page



How AJAX works

1. An event occurs in a web page (the page is loaded, a button is clicked)
2. An XMLHttpRequest object is created by JavaScript
3. The XMLHttpRequest object sends a request to a web server
4. The server processes the request
5. The server sends a response back to the web page
6. The response is read by JavaScript
7. Proper action (like page update) is performed by JavaScript

Asynchronous Programming with JavaScript

Overview of AJAX



The XMLHttpRequest Object

Configuring an AJAX Request

Handling Errors on readystatechange

Introduction to Promises

Creating the XMLHttpRequest Object

- Fairly straight-forward, unless need to support old IE browsers

```
16  var xmlhttp;
17  if (window.XMLHttpRequest) {
18      xmlhttp = new XMLHttpRequest();
19  } else {
20      // code for IE6, IE5
21      xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");
22  }
```

Working with the XMLHttpRequest Object

- Setup a function called **onreadystatechange**
 - it will be called when readyState changes

```
29  xmlhttp.onreadystatechange = function() {  
30  if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {  
31      var myArr = JSON.parse(xmlhttp.responseText);  
32      myFunction(myArr);  
33  }  
34 };
```

Specify and Send the request

- open: the request type, the URL and a boolean to setup an asynch request
- send: sends the request to the server

```
19 xmlhttp.onreadystatechange = function() {
20 if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
21     var myArr = JSON.parse(xmlhttp.responseText);
22     myFunction(myArr);
23 }
24 };
25
26 xmlhttp.open("GET", url, true);
27 xmlhttp.send();
```

Properties of XHR

Property	Description
onreadystatechange	Defines function to be called when readyState property changes
readyState Holds the status of the XMLHttpRequest.	0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
status	200: "OK" 403: "Forbidden" 404: "Page not found"
statusText	Returns the status-text (e.g. "OK" or "Not Found")

Asynchronous Programming with JavaScript

The XMLHttpRequest Object



Configuring an AJAX Request

Handling Errors on readystatechange

Introduction to Promises

Setting up the request: `open()` and `send()`

Method	Description
<code>open(method, url, async)</code> (Specifies the type of request)	<p><i>method</i>: the type of request: GET or POST</p> <p><i>url</i>: the server (file) location</p> <p><i>async</i>: true (asynchronous) or false (synchronous)</p>
<code>send()</code>	Sends the request to the server (used for GET)
<code>send(string)</code>	Sends the request to the server (used for POST)

AJAX Request

- The AJAX request requires several operations
 - Create a handler function to receive the asynchronous response
 - Make the open call with the method and URL
 - Send the request

```
15  var xmlhttp = new XMLHttpRequest();
16  var url = "data/myBands.json";
17
18  xmlhttp.onreadystatechange = function() {
19    if (xmlhttp.readyState == 4 && xmlhttp.status == 200) {
20      var myArr = JSON.parse(xmlhttp.responseText);
21      myFunction(myArr);
22    }
23  };
24
25  xmlhttp.open("GET", url, true);
26  xmlhttp.send();
```

AJAX Response

- When the response comes back the function is called
 - It then updates the DOM

```
28  function myFunction(jsonData) {  
29      var out = "";  
30      var bands = jsonData.bands.band;  
31      var i;  
32      for(i = 0; i < bands.length; i++) {  
33          out += bands[i].name + ' was formed ' + bands[i].yearFormed + ' in ' + bands[i].city + '<br/>';  
34      }  
35      document.getElementById("showBands").innerHTML = out;  
36  }
```

Asynchronous Programming with JavaScript

The XMLHttpRequest Object

Configuring an AJAX Request



Handling Errors on readystatechange

Introduction to Promises

Handling Errors

- Use readyState to call during a certain stage of loading
- Perform different actions for the different statuses

```
5
6     xhr.onreadystatechange = function () {
7         if (xhr.readyState == 2 && xhr.status == 404) {
8             document.getElementById("showBands").innerHTML = 'Unable to access database';
9         }
10
11        if (xhr.readyState === 4 && xhr.status == 200) {
12            var myArr = JSON.parse(xhr.responseText);
13            bandsData(myArr);
14        }
15    };
16
```

Asynchronous Programming with JavaScript

The XMLHttpRequest Object

Configuring an AJAX Request

Handling Errors on readystatechange



Introduction to Promises

Callback Hell

```
step1(function (value1) {  
    step2(value1, function(value2) {  
        step3(value2, function(value3) {  
            step4(value3, function(value4) {  
                // Do something with value4  
            });  
        });  
    });  
});  
});
```

A Promise library can flatten the structure

- One benefit is improving multiple nested calls to async functions

With more intuitive API

More at

<http://www.html5rocks.com/en/tutorials/es6/promises/>

```
start()  
.then(promisedStep1)  
.then(promisedStep2)  
.then(promisedStep3)  
.then(function (value4) {  
    // Do something with value4  
});
```

Besides flattening callback hell structure...

- More robust error handling
- Composable operations
- Unifying synchronous and asynchronous result/error channels
- Easier management of concurrent operations
 - Promise.all
 - Promise.race
 - Promise.map

Different Libraries Offer Different Promise APIs

- ES6 has incorporated Promises
 - Barebones, other libraries have more advantages
 - ES6 Promises lack standardized debugging
 - Are missing essential utilities such Promise.try/promisify/promisifyAll
 - Cannot catch specific error types
- Q
 - <https://github.com/kriskowal/q>
- When.js
 - <https://github.com/cujojs/when>

Bluebird



- Bluebird
 - <http://bluebirdjs.com/docs/anti-patterns.html>
- Better error handling (because of error filtering)
- Better debugging (long stacktraces, unReturned Promise warnings, ...)
- Ships with `promisify/promisifyAll` implementations for converting existing error-first callback APIs to Promise-returning APIs
- Ships with a number of important utilities such as `Promise.map`
- More robust unhandled error handling
- Events are consistently implemented between runtimes - Unlike ES6

Chapter Summary

In this chapter we have:

- Made asynchronous calls to a server using JS
- Configured AJAX requests
- Handled errors
- Introduced Promises

Introduction to Modern JavaScript (ES6+)

Course Summary

- During this course we have covered:
 - Store information in variables and objects
 - Create code blocks for conditional logic and looping in code
 - Utilize browser developer tools for debugging JavaScript code
 - Create event handlers to respond to browser events
 - Create and call functions with parameters
 - Parse and create JSON objects
 - Create browser side scripts for manipulating the Document Object Model (DOM)
 - Create asynchronous calls to a Web server using AJAX
 - Discuss code organization techniques, including modules