

## CHAPTER 4

# Introduction to JavaScript

Pralhad Kumar Shrestha

# Contents:

1. Basics of JavaScript and Document Object Model
2. Element Access in Java scripts, event and event handling
3. DOM Event Model and Element Positioning
4. Moving elements and Element visibility
5. Changing colors and fonts
6. Dynamic content and stacking elements
7. Locating the mouse cursor and reacting to a mouse click
8. Dragging and dropping elements

# 1. Basics of JavaScript and DOM

## Origins of JavaScript

- Originally developed by Netscape, as LiveScript.
- Became a joint venture of Netscape and Sun in 1995, renamed to JavaScript (JS).
- Standardized by the European Computer Manufacturers Association (ECMA) as **ECMA-262**.
- Official name of the standard language is **ECMAScript**.
- An HTML-embedded scripting language.
- We call collections of JavaScript code scripts, not programs.

# 1. Basics of JavaScript and DOM

- JavaScript (JS) is a lightweight, interpreted programming language.
- It is mostly known as the scripting language for Web pages.
- JavaScript is an interpreted language.
- JavaScript was designed to add interactivity and programming to HTML web pages.
  - Performs calculations such as totaling the price or computing sales tax.
  - Verifies data without sending to the server.
  - Adapts the display to user needs.
- JavaScript is case-sensitive.

# 1. Basics of JavaScript and DOM

- JavaScript is object-based language as it provides predefined objects.
- JavaScript is both dynamically and loosely typed language.
  - "Loosely typed" means language does not bother with types too much, and does conversions automatically.
  - In dynamic typing, types are associated with values not variables.
  - "Dynamically typed language" => you do not need to declare variable type before use.

# JavaScript and Java

- JavaScript and Java are only related through syntax (expressions, assignment statements and control statements).
- JavaScript is dynamically typed.
- JavaScript's support for objects is very different.
- JavaScript is interpreted.
  - Source code is embedded inside HTML doc, there is no compilation

# Uses of JavaScript

- Transfer of some load from server to client.
- User interactions through forms
  - Events easily detected with JavaScript
  - E.g. validate user input
- The Document Object Model makes it possible to create dynamic HTML documents with JavaScript.
- JavaScript scripts are executed entirely by the browser.
- Once downloaded JS codes cannot be modified from the server until page is refreshed.
  - But HTTP request can be invoked by JS to exchange the information with the server.

# Object Orientation of JavaScript

- JavaScript is NOT an object-oriented programming language.
  - It is an object-based programming language.
- Does not support class-based inheritance.
  - Cannot support polymorphism.
- JavaScript objects are collections of properties, which are like the members of classes in Java.
  - Data and method properties
- JavaScript has primitives for simple types.
- The root object in JavaScript is Object
  - all objects are derived from Object.



# Embedding JavaScript

- JavaScript scripts are embedded, either directly or indirectly, in HTML/XHTML documents.

## 1. Internal JavaScript:

- ❖ Added inside the body of HTML document.
- ❖ When written within the html element using attributes related to events of the element.
- ❖ Can be added into **head** or **body** section depending on when you want the JS code to load.

```
<script>  
    var date = new Date();  
    alert("Today's date is " + date);  
</script>
```

# Embedding JavaScript

- JavaScript scripts are embedded, either directly or indirectly, in HTML/XHTML documents.

## 1. Internal JavaScript:

- ❖ Added inside the script tag of HTML document.
- ❖ Can be added into **head** or **body** section depending on when you want the JS code to load.

```
<script>  
    var date = new Date();  
    alert("Today's date is " + date);  
</script>
```

# Embedding JavaScript

- ❖ The above code when added inside either of head or body tag, results in same value.
- ❖ The below code won't work if added into head tag.

```
<script>  
document.getElementById("demo").innerHTML = "My First JavaScript";  
</script>
```

# Embedding JavaScript

```
<!DOCTYPE html>
<html>
<body>
<h2>JavaScript in Body</h2>
<p id="demo"></p>
<script type="text/javascript">
document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
</body>
</html>
```

**JavaScript in Body**

My First JavaScript

# Embedding JavaScript

```
<!DOCTYPE html>
<html>
<head>
<script type="text/javascript">
document.getElementById("demo").innerHTML = "My First JavaScript";
</script>
</head>
<body>
<h2>JavaScript in Body</h2>
<p id="demo"></p>
</body>
</html>
```

**JavaScript in Body**

# Embedding JavaScript

## 2. External JavaScript:

- ❖ Code is written into separate file with .js extension.
- ❖ Can be added into **head** or **body** section depending on when you want the JS code to load.
- ❖ Code is included using script tag's **src** attribute.

```
<script type="text/javascript" src="myScript.js"></script>
```

myScript.js

```
function myFunction()  
{  
  document.getElementById("demo").innerHTML="Paragraph changed.";  
}
```

# Embedding JavaScript

- ❖ HTML and JS are easier to read and maintain.
- ❖ Cached JS files can speed up page load.

## 3. Inline JavaScript:

- ❖ Added inside the body of HTML document.
- ❖ Written within the html element using attributes related to events of the element.

```
<button type="button" onclick="myFunction()">Try it</button>
```

# Embedding JavaScript

```
<!DOCTYPE html>
<html>
<body>
<h2>Demo JavaScript in Body</h2>
<p id="demo">A Paragraph</p>
```

```
<button type="button" onclick="myFunction()">Try it</button>
```

```
<script>
function myFunction()
{
  document.getElementById("demo").innerHTML = "Paragraph changed.";
}
</script>
```

```
</body>
</html>
```




# General Syntactic Characteristics

- Identifier form: begin with a letter or underscore, followed by any number of letters, underscores, and digits
  - Case sensitive
- 25 reserved words, plus future reserved words
- Comments: both `//` and `/* ... */`
- Scripts are usually hidden from browsers that do not include JavaScript interpreters by putting them in special comments

```
<!--  
    -- JavaScript script --  
//-->
```

# General Syntactic Characteristics

- Semicolons can be a problem
  - They are “somewhat” optional.
  - Problem: When the end of the line can be the end of a statement – JavaScript puts a semicolon there



```
return  
x;
```

- Here JS adds semicolon after return and makes x an illegal orphan.
- The primitive data types, operations, and expressions of JavaScript are similar to other common programming language.

# Primitives, Operations and Expressions

- JavaScript has five primitive types:
  - Number
  - String
  - Boolean
  - Undefined
  - Null
- A Boolean represents only one of two values:
  - true, or false

var positive = **true**;

var negative = **false**;

# Primitives, Operations and Expressions

- There is only one type of Number in JavaScript.
- Numbers can be written with or without a decimal point.  
`var decimalVal = 75.00; // with decimals`  
`var intVal = 75; // without decimals`
- Variable without a value will have **undefined** value.  
`var data; => value of data is undefined`
- **null** => "nothing" (null is supposed to be something that doesn't exist)
- Strings are written with quotes.

# Primitives, Operations and Expressions

- We can use single or double quotes:  
    `var stringOne = "Volvo XC60"; // double quote is used`  
    `var stringTwo = 'Volvo XC60'; // single quote is used`
- JavaScript is dynamically typed. (Same variable can be used to hold different data types. Also, can reference to different object.)  
    `var dynamicVariable = 5; //numeric`  
    `var dynamicVariable = "scorpio"; //string`  
    `var dynamicVariable = true; //boolean`

# Declaring Variables

- The interpreter determines the type of a particular occurrence of a variable.
- Variables can be either implicitly or explicitly declared.

```
var sum = 0,  
    counter,  
    today = "Monday",  
    flag = false;
```

Explicit declaration

```
<script>  
counter = 10;  
console.log(counter);  
</script>
```

Implicit declaration

- **Recommended method:** declare all variables explicitly.

# Numeric Operators

Operator	Associativity
++, --, unary -	Right (though it is irrelevant)
*, /, %	Left
Binary +, binary -	Left

The first operators listed have the highest precedence.

Learn More at:

[https://www.w3schools.com/js/js\\_operators.asp](https://www.w3schools.com/js/js_operators.asp)

# Numeric Operators

```
var a = 2, b = 4, c, d;
```

```
c = 3 + a * b; // * is first, so c is now 11 (not 24)
```

```
d = b / a / 2; // / associates left, so d is now 1 (not 4)
```

```
(a + b) * c; // addition will be done before multiplication
```



# The Math and Number Object

- The *Math* Object provides *floor*, *round*, *max*, *min*, trig functions, etc.
  - e.g. `Math.cos(x)`
- The *Number* Object has some useful properties:

Property	Meaning
<code>MAX_VALUE</code>	Largest representable number
<code>MIN_VALUE</code>	Smallest representable number
<code>NaN</code>	Not a number
<code>POSITIVE_INFINITY</code>	Special value to represent infinity
<code>NEGATIVE_INFINITY</code>	Special value to represent negative infinity
<code>PI</code>	The value of $\pi$

Learn More at: [https://www.w3schools.com/js/js\\_math.asp](https://www.w3schools.com/js/js_math.asp)  
[https://www.w3schools.com/js/js\\_numbers.asp](https://www.w3schools.com/js/js_numbers.asp)

# The String Object

- The number of characters in a string is stored in the length property

```
var str = "George";  
var len = str.length; //6
```

Learn More at:

[https://www.w3schools.com/js/js\\_strings.asp](https://www.w3schools.com/js/js_strings.asp)

- Common string methods:

Method	Parameters	Result
<code>charAt</code>	A number	The character in the <code>String</code> object that is at the specified position
<code>indexOf</code>	One-character string	The position in the <code>String</code> object of the parameter
<code>substring</code>	Two numbers	The substring of the <code>String</code> object from the first parameter position to the second
<code>toLowerCase</code>	None	Converts any uppercase letters in the string to lowercase
<code>toUpperCase</code>	None	Converts any lowercase letters in the string to uppercase

# The Date Object

- Create one with the Date constructor (without any parameters)

```
var date = new Date();
```

Learn More at:

[https://www.w3schools.com/js/js\\_dates.asp](https://www.w3schools.com/js/js_dates.asp)

- Local time methods of Date:

- toLocaleString – returns a string of the date
- getDate – returns the day of the month
- getMonth – returns the month of the year (0 – 11)
- getDay – returns the day of the week (0 – 6)
- getFullYear – returns the year
- getTime – returns the number of milliseconds since January 1, 1970
- getHours – returns the hour (0 – 23)
- getMinutes – returns the minutes (0 – 59)
- getMilliseconds – returns the millisecond (0 – 999)

# The typeof Operator

- The typeof operator returns the type of single operand.  
typeof(myVariable)

```
var myVariable = 100;  
var myString = "string";  
  
typeof(myVariable);  
typeof(myString);
```

# JavaScript Type Conversion

- JavaScript variables can be converted to a new variable and another data type:
  - By the use of a JavaScript function (Explicit)
  - Automatically by JavaScript itself (Implicit)

```
Number("3.14") // returns 3.14
Number("")      // returns 0
Number(true)    // returns 1
String(x)       // returns a string from a number variable x
String(123)     // returns a string from a number literal 123
String(false)  // returns "false"
```

```
var y = "5";    // y is a string
var x = + y;    // x is a number
```

# JavaScript Type Conversion

```
// if myVar = 123           // toString converts to "123"  
// if myVar = true         // toString converts to "true"  
// if myVar = false        // toString converts to "false"  
// if myVar = {name:"Fjohn"} // toString converts to "[object Object]"  
// if myVar = [1,2,3,4]    // toString converts to "1,2,3,4"  
// if myVar = new Date()   // toString converts to "Fri Jul 18 2014 09:08:55 GMT+0200"
```

```
document.getElementById("demo").innerHTML = myVar;
```

## Automatic String Conversion

# Screen Output And Keyboard Input

- JavaScript models the HTML document with the *Document* object.
- The model for the browser display window is the *Window* object.
  - The *Window* object has two properties, *document* and *window*, which refer to the *Document* and *Window* objects, respectively.
- The *Document* object has a method, *write*, which dynamically creates content.
  - The parameter is a string, often concatenated from parts, some of which are variables.

```
document.write("Answer: ", result, "<br>");
```

- The parameter is sent to the browser, so it can be anything that can appear in an HTML document (any HTML tags).

# Screen Output And Keyboard Input

- The *Window* object has three methods for creating dialog boxes

## 1. Alert

- Parameter is plain text, not HTML.
- Opens a dialog box which displays the parameter string and an OK button.

```
alert("The sum is:" + sum + "\n");
```





# Screen Output And Keyboard Input

## 2. Confirm

- Opens a dialog box and displays the parameter and two buttons, OK and Cancel.
- Returns a Boolean value, depending on which button was pressed (it waits for one).

```
var question = confirm("Do you want to continue this download?");
```

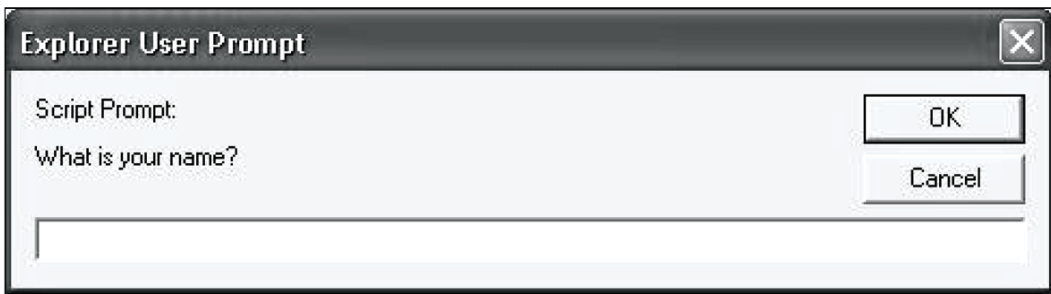


# Screen Output And Keyboard Input

## 3. Prompt

- Opens a dialog box and displays its string parameter, along with a text box and two buttons, OK and Cancel.
- The second parameter is for a default response if the user presses OK without typing a response in the text box (waits for OK).

```
prompt("What is your name?", " ");
```



# Screen Output And Keyboard Input

```
<?xml version = "1.0"?>  
<!DOCTYPE html PUBLIC "-//w3c//DTD XHTML 1.1//EN"  
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtda">
```

```
<!-- roots.html  
A document for roots. js  
-->  
<html xmlns = "http://www.w3.0rg/1999/xhtml11">  
  <head>  
    <title> roots.html </title>  
  </head>  
  <body>  
    <script type = "text/javascript" src = "roots.js">  
    </script>  
  </body>  
</html>
```

# Screen Output And Keyboard Input

```
// roots.js
//Compute the real roots of a given quadratic equation. If the roots are imaginary, this script displays
//Nan, because that is what results from taking the square root of a negative number

// Get the coefficients of the equation from the user
var a = prompt ("What is the value of 'a'? \n", " ");
var b = prompt("What is the value of 'b'? \n", " ");
var c = prompt("What is the value of 'c'? \n", " ");

// Compute the square root and denominator of the result
var root_part = Math.sqrt(b* b - 4.0 * a * C);
var denom = 2.0 * a;

// Compute and display the two roots
var root1 = (-b + root part) / denom;
var root2 = (-b - root part) / denom;
document.write("The first root is: ", root1, "<br />");
document.write("The second root is: ", root2, "<br />");
```

# Control Statements

- Control statements often require some syntactic container for sequences of statements whose execution they are meant to control.
- Compound statement in JS is a sequence of statements delimited by *braces*.
  - Container is a compound statement in JS
- Control construct is a control statement and the statement or compound statement whose execution it controls.

# Control Statements

- Selection Statements (if... else if... else)
- Switch Statements
- Loop Statements (for, while, do.. while)

# Object creation and modification

- Objects are created with a new expression, which must include a call to a constructor method.

```
var car = new Object();  
car.name = "Volkswagon";  
car.model = "X300";  
car.year = 2020;
```

```
console.log(car.name); //Volkswagon  
console.log(car.model); //X300  
console.log(car.year); //2020
```

```
car.year = 2022;  
console.log(car.year); //2022
```

```
console.log(car);  
  
//car object output  
{  
  name: "Volkswagon",  
  model: "X300",  
  year: 2020  
}
```

# Arrays

- Array elements can be primitive values or references to other objects.
- Array objects can be created in two ways, with `new`, or by assigning an array literal.

```
var myList = new Array(24, "bread", true);
```

```
var myList2 = new Array(24);
```

```
var myList3 = [24, "bread", true];
```

- Length is **dynamic** - the *length* property stores the length.
- `length` property is writeable.  

```
myList.length = 150;
```



# Array Methods

<code>toString()</code>	converts an array to a string of (comma separated) array values.
<code>join()</code>	joins all array elements into a string.
<code>pop()</code>	removes the last element from an array.
<code>push()</code>	adds a new element to an array (at the end).
<code>shift()</code>	removes the first array element and "shifts" all other elements to a lower index.
<code>unshift()</code>	adds a new element to an array (at the beginning), and "unshifts" older elements.
<code>concat()</code>	creates a new array by merging (concatenating) existing arrays

# Functions

```
function function_name([formal_parameters]) {  
  -- body --  
}
```

- Return value is the parameter of return
  - If there is no return or if return has no parameter, undefined is returned
- We place all function definitions in the head of the HTML document
  - Calls to functions appear in the document body
- Variables explicitly declared in a function are local

# Functions

- Parameters are passed by value, but when a reference variable is passed, the semantics are pass-by-reference
- There is no type checking of parameters, nor is the number of parameters checked
  - excess actual parameters are ignored, excess formal parameters are set to undefined
- All parameters are sent through a property array, *arguments*, which has the *length* property

# Constructors

- Constructors are special methods that creates and initialize the properties for newly created objects.
- Every new expression must include a call to a constructor, whose name is the same as the object being created.

```
function car(newName, newModel, newYear)
{
    this.name = newName;
    this.model = newModel;
    this.year = newYear;
}
```

```
//create a new instance of car
myCar = new car("Ford", "Santa SUV", 2021);
```

# Pattern Matching using RegEx

- There are two approaches to pattern matching in JavaScript: one that is based on the RegExp object and one that is based on methods of the String object.
- The regular expressions used by these two approaches are the same.
- Regular Expressions, commonly known as "regex" or "RegExp", are a specially formatted text strings used to find patterns in text.
  - Syntax: /pattern/modifiers;
- It can be used to verify whether the format of data i.e. name, email, phone number, etc. entered by the user is correct or not, find or replace matching string within text content etc.

# Pattern Matching using RegEx

- JavaScript's built-in methods for performing pattern-matching
  - Search()
    - The simplest pattern-matching method is search, which takes a regular expression as a parameter.
      - `string.search(RegExp);` 7
    - The search method returns the index of the first match in the String object (through which it is called).
    - If there is no match, search returns `'-1'`.

```
<script>  
  var str = "Please locate where 'locate' occurs!";  
  document.getElementById("demo").innerHTML = str.search("locate");  
</script>
```

//returns 7

# Pattern Matching using RegEx

## 2. Replace()

- The replace() method returns a modified string where the pattern is replaced.
- The replace method is used to replace substrings of the String object that match the given pattern
  - `string.replace(searchvalue, newvalue);`

```
<script>  
  var text = "Visit Microsoft!";  
  var result = text.replace("Microsoft", "W3Schools");  
</script>
```

```
//returns  
Visit W3Schools!
```

# Pattern Matching using RegEx

## 3. Match()

- Search for a match in a string. It returns an array of information or null on mismatch.
  - `string.match(regex);`

```
<script>
  var text = "The rain in SPAIN stays mainly in the plain";
  var result = text.match(/ain/);

  document.getElementById("demo").innerHTML = result;
</script>
```

//returns ain



# Character and Character-Class Patterns

- **Metacharacters** are characters that have special meanings in some contexts in patterns.
- The “normal” characters are those that are not metacharacters.
- The pattern metacharacters are:

`\ | ( ) [ ] { } ^ $ * + ? .`

- Metacharacters can themselves be matched by being immediately preceded by a backslash.
- Square brackets surrounding a pattern of characters are called a character class.

e.g. `[abcde]`.

```
<script>
  var text = "Visit W3Schools";
  var result = text.match(/w3schools/i);
  console.log(result);
</script>
```

# Character and Character-Class Patterns

- Modifiers can be used to perform case-insensitive more global searches:

Expression	Description
i	Perform case-insensitive matching
g	Perform a global match (find all matches rather than stopping after the first match)
m	Perform multiline matching

# Character and Character-Class Patterns

- Brackets are used to find a range of characters:

Expression	Description
[abc]	Find any of the characters between the brackets. i.e. (a, b, c, ac, bc)
[a-z]	Finds the lowercase characters from a to z.
[A-Z]	Finds the uppercase characters from A to Z.
[a-h]	Finds the lowercase characters from a to h.
[0-9]	Find any of the digits between the brackets.
(x y)	Find any of the alternatives separated with
[^aeiou]	With circumflex character(^) in the beginning, it finds any characters except a, e, i, o, u.

# Character and Character-Class Patterns

- Predefined character classes:

Name	Equivalent Pattern	Matches
<code>\d</code>	<code>[0-9]</code>	A digit
<code>\D</code>	<code>[^0-9]</code>	Not a digit
<code>\w</code>	<code>[A-Za-z_0-9]</code>	A word character (alphanumeric)
<code>\W</code>	<code>[^A-Za-z_0-9]</code>	Not a word character
<code>\s</code>	<code>[ \r\t\n\f]</code>	A whitespace character
<code>\S</code>	<code>[^ \r\t\n\f]</code>	Not a whitespace character

# Character and Character-Class Patterns

- Quantifiers defines quantity to solve various of problems:

Expression	Description
<code>n+</code>	Matches any string that contains at least one <code>n</code> .
<code>n*</code>	Matches any string that contains zero or more occurrences of <code>n</code> .
<code>n?</code>	Matches any string that contains zero or one occurrences of <code>n</code> .
<code>n{3}</code>	Matches exactly 3 occurrences of letter <code>n</code> .
<code>n{5, 10}</code>	Matches at least 5 occurrences and not more than 10 occurrences of letter <code>n</code> .
<code>n{3,}</code>	Matches 3 or more occurrences of letter <code>n</code> .
<code>n{, 3}</code>	Matches at most 3 occurrences of letter <code>n</code> .

# Character and Character-Class Patterns

- Symbolic quantifiers
  - An asterisk means zero or more repetitions.
  - A plus sign means one or more repetitions.
  - A question mark means one or none.

# Character and Character-Class Patterns

```
var phoneRegex = /^[7-9][0-9]{9}$/;  
var phone = 9806612345;  
phoneRegex.test(phone);
```

```
var emailRegex = /^\\w+([\\.-]?\\w+)*@\\w+([\\.-]?\\w+)*\\.\\w{2,3}+$/;  
var email = “myemail@gmail.com”;  
emailRegex.test(email);
```

# JavaScript Execution Environment

- The Window object provides the largest enclosing referencing environment for scripts
- Implicitly defined Window properties:
  - **document** - a reference to the Document object that the window displays
  - **frames** - an array of references to the frames of the document
- Every Document object has:
  - **forms** - an array of references to the forms of the document
    - Each *forms* object has an *elements* array, which has references to the form's elements
  - *Document* also has property arrays for anchors, links, & images



# The Document Object Model

- Document Object Model(DOM) 0 is supported by all JavaScript-enabled browsers (no written specification)
- DOM 1 was released in 1998
- DOM 2 issued in 2000
  - Nearly completely supported by NS7
  - IE6's support is lacking some important things
- DOM 3 is the latest W3C specification
- The DOM is an abstract model that defines the interface between HTML documents and application programs—an API

# The Document Object Model

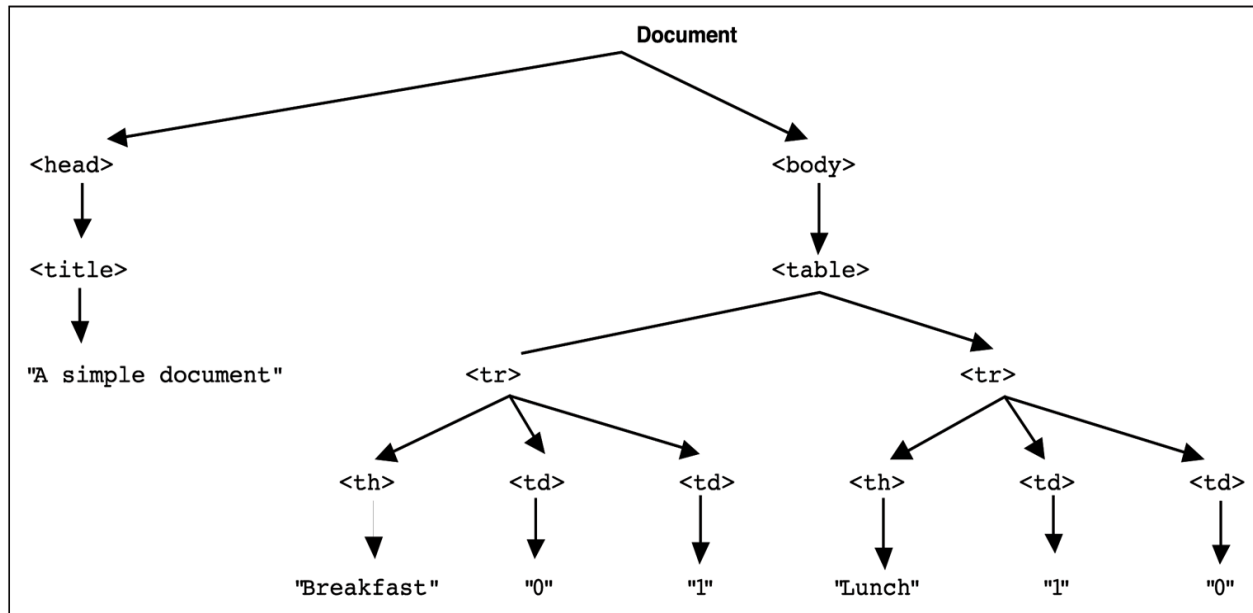
- A language that supports the DOM must have a binding to the DOM constructs
- In the JavaScript binding, HTML elements are represented as objects and element attributes are represented as properties
- e.g., `<input type = "text" name = "address">`
  - would be represented as an object with two properties, type and name, with the values "text" and "address"

# Document Object Model Structure

- Documents in the DOM have a tree like structure

# Document Object Model Structure

```
<html>
  <head> <title> A simple document </title>
</head>
<body>
  <table>
    <tr>
      <th>Breakfast</th>
      <td>0</td>
      <td>1</td>
    </tr>
    <tr>
      <th>Lunch</th>
      <td>1</td>
      <td>0</td>
    </tr>
  </table>
</body>
</html>
```



# Element Access in JavaScript

- There are several ways to do it
  1. DOM Address
    - Example (a document with just one form and one widget):  
`<form action = ">`  
    `<input type = "button" name = "pushMe">`  
`</form>`
    - Can be accessed using => `document.forms[0].elements[0]`
    - Problem: document changes

# Element Access in JavaScript

## 2. Element names

- requires the element and all of its ancestors (except body) to have `name` attributes

```
<form name = "myForm" action = "">
```

```
  <input type = "button" name = "pushMe">
```

```
</form>
```

- Can be accessed using `=> document.myForm.pushMe`

# Element Access in JavaScript

## 3. getElementById Method

- Example (a document with just one form and one widget):

```
<form action = "">
```

```
  <input type = "button" name = "pushMe">
```

```
</form>
```

- Can be accessed using => `document.getElementById("pushMe")`
- Form elements often have ids and names both set to the same value

# Element Access in JavaScript

- Checkboxes and radio button have an implicit array, which has their name.

```
<form id = "toppingGroup">
  <input type = "checkbox" name = "toppings"
    value = "olives" />
  <input type = "checkbox" name = "toppings"
    value = "tomatoes" />
</form>

var numChecked = 0;
var dom = document.getElementById("toppingGroup");
for (index = 0; index < dom.toppings.length; index++)
  if (dom.toppings[index].checked)
    numChecked++;
```



# Events and Event Handling

- An ***event*** is a notification that something specific has occurred, either with the browser or an action of the browser user.
- An ***event handler*** is a script that is implicitly executed in response to the appearance of an event.
- The process of connecting an event handler to an event is called ***registration***.
- Don't use *document.write* in an event handler, because the output may go on top of the display.

# Events and their Tag Attributes

## Event

blur

change

click

focus

load

mousedown

mousemove

mouseout

mouseover

mouseup

select

submit

unload

## Tag Attribute

onblur

onchange

onclick

onfocus

onload

onmousedown

onmousemove

onmouseout

onmouseover

onmouseup

onselect

onsubmit

onunload

# Events, Attributes and Tags

- The same attribute can appear in several different tags
  - e.g., The *onclick* attribute can be in `<a>` and `<input>`
- A text element gets focus in three ways:
  - When the user puts the mouse cursor over it and presses the left button
  - When the user tabs to the element
  - By executing the *focus* method

# Events, Attributes and Tags

Attribute	Tag	Description
onblur	<a>	The link loses the input focus.
	<button>	The button loses the input focus.
	<input>	The input element loses the input focus.
	<textarea>	The text area loses the input focus.
	<select>	The selection element loses the input focus.
onchange	<input>	The input element is changed and loses the input focus.

	<code>&lt;textarea&gt;</code>	The text area is changed and loses the input focus.
	<code>&lt;select&gt;</code>	The selection element is changed and loses the input focus.
<code>onclick</code>	<code>&lt;a&gt;</code>	The user clicks on the link.
	<code>&lt;input&gt;</code>	The input element is clicked.
<code>onfocus</code>	<code>&lt;a&gt;</code>	The link acquires the input focus.
	<code>&lt;input&gt;</code>	The input element receives the input focus.
	<code>&lt;textarea&gt;</code>	A text area receives the input focus.
	<code>&lt;select&gt;</code>	A selection element receives the input focus.
<code>onload</code>	<code>&lt;body&gt;</code>	The document is finished loading.
<code>onmousedown</code>	Most elements	The user clicks the left mouse button.
<code>onmousemove</code>	Most elements	The user moves the mouse cursor within the element.
<code>onmouseout</code>	Most elements	The mouse cursor is moved away from being over the element.
<code>onmouseover</code>	Most elements	The mouse cursor is moved over the element.
<code>onmouseup</code>	Most elements	The left mouse button is unclicked.
<code>onselect</code>	<code>&lt;input&gt;</code>	The mouse cursor is moved over the element.
	<code>&lt;textarea&gt;</code>	The text area is selected within the text area.
<code>onsubmit</code>	<code>&lt;form&gt;</code>	The Submit button is pressed.
<code>onunload</code>	<code>&lt;body&gt;</code>	The user exits the document.

# Registration of Event Handler

- By assigning the event handler script to an event tag attribute.

```
<input type "button" name = "myButton" onclick = "alert('Mouse  
click!');" />
```

```
<input type "button" name = "myButton"          onclick = "myHandler();" />
```

# Handling Events from Body Elements

- Events most often created by body elements are *load* and *unload*
- Example:
  - The *load* event - triggered when the loading of a document is completed

**Check the examples for all other topics**



# End of chapter 4