COMP519 Web Programming

Lecture 17: JavaScript (Part 8)

Handouts

Ullrich Hustadt

Department of Computer Science
School of Electrical Engineering, Electronics, and Computer Science
University of Liverpool

Contents

Window and Document objects

JavaScript provides two objects that are essential to the creation of dynamic web pages and interactive web applications:

document object

- an object-oriented representation of a web page (HTML document) that is displayed in a window
- allows interaction with the Document Object Model (DOM) of a page

  Example: `document.writeln()` adds content to a web page

Document Object Model

A platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of HTML, XHTML and XML documents

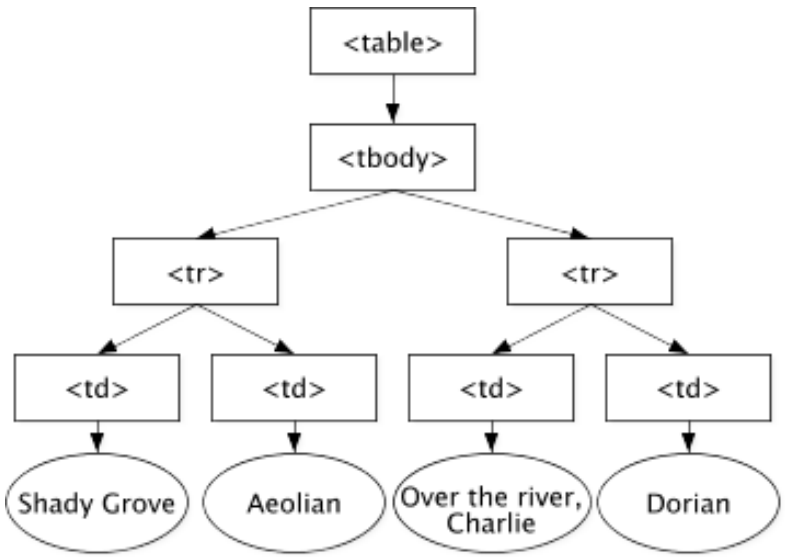
Document Object Model

Example:

The HTML table below

```
<table>
  <tbody>
    <tr>
      <td>Shady Grove</td>
      <td>Aeolian</td>
    </tr>
    <tr>
      <td>Over the River, Charlie</td>
      <td>Dorian</td>
    </tr>
  </tbody>
</table>
```

is parsed into the following DOM



Arnaud Le Hors, et al, editors: Document Object Model (DOM) Level 3 Core Specification, Version 1.0, W3C Recommendation 07 April 2004. World Wide Web Consortium, 2004. https://www.w3.org/TR/DOM-Level-3-Core/ [accessed 9 January 2017]

Accessing HTML Elements: Object Methods

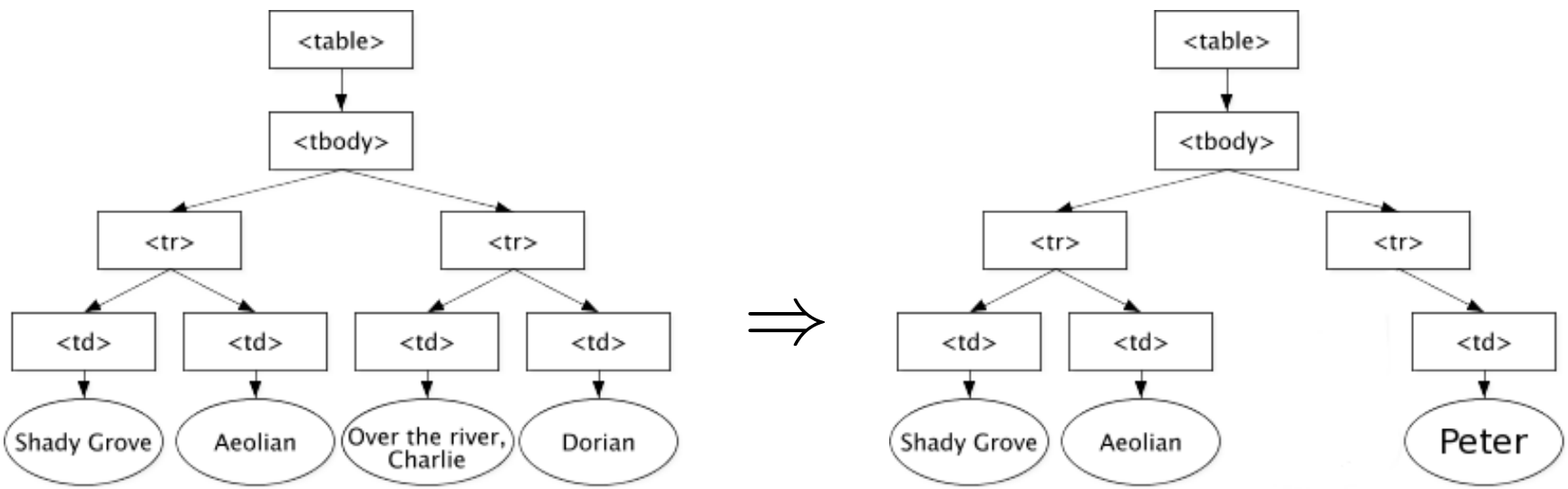
Example:

```
// access the tbody element from the table element
var myTbodyElement = myTableElement.firstChild;

// access its second tr element; the list of children starts at 0 (not 1).
var mySecondTrElement = myTbodyElement.childNodes[1];

// remove its first td element
mySecondTrElement.removeChild(mySecondTrElement.firstChild);

// change the text content of the remaining td element
mySecondTrElement.firstChild.firstChild.data = "Peter";
```



Accessing HTML Elements: Names (1)

Instead of using methods such as `firstChild` and `childNodes[n]`, it is possible to assign names to denote the children of an HTML element

Example:

```
<form name="form1" action="">
<label>Temperature in Fahrenheit:</label>
<input type="text" name="fahrenheit" size="10" value="0"><br>
<label>Temperature in Celsius:</label>
<input type="text" name="celsius"    size="10" value="">
</form>
```

Then – `document.form1`
Refers to the form named `form1`

– `document.form1.celsius`
Refers to the text field named `celsius` in `document.form1`

– `document.form1.celsius.value`
Refers to the attribute `value` in the text field named `celsius` in `document.form1`

Accessing HTML elements: Names (2)

Accessing HTML elements by giving them names and using paths within the Document Object Model tree structure is still problematic

⇝ If that tree structure changes, then those paths no longer work

Example:

Changing the previous form to

```
<form name="form1" action="">
<div class="field" name="fdiv">
<label>Temperature in Fahrenheit:</label>
<input type="text" name="fahrenheit" size="10" value="0">
</div>
<div class="field" name="cdiv">
<label>Temperature in Celsius:</label>
<input type="text" name="celsius"    size="10" value="" >
</div>
</form>
```

means that `document.form1.celsius` no longer works as there is now a `div` element between form and text field, we would now need to use `document.form1.cdiv.celsius`

Accessing HTML elements: IDs

A more reliable way is to give each HTML element an ID (using the `id` attribute) and to use `getElementById` to retrieve an HTML element by its ID

Example:

```
<form id="form1" action="">
Temperature in Fahrenheit:
<input type="text" id="fahrenheit"  size="10" value="0"><br>
Temperature in Celsius:
<input type="text" id="celsius"     size="10" value="" ><br>
</form>
```

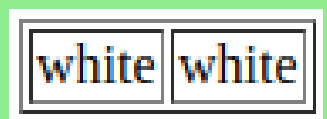
Then

- `document.getElementById('celsius')`
  Refers to the HTML element with ID `celsius` `document`
- `document.getElementById('celsius').value`
  Refers to the attribute `value` in the HTML element with ID `celsius` in `document`

Manipulating HTML elements (1)

It is not only possible to access HTML elements,
but also possible to change them on-the-fly

```
<html><head><title>Manipulating HTML elements (1)</title>
<style>
  td.RedBG { background: #f00; }
</style>
<script>
function changeBackgroundBlue(id) {
  document.getElementById(id).style.background = "#00f";
  document.getElementById(id).innerHTML = "blue";
}
function changeBackgroundRed(cell) {
  cell.className = "RedBG";
  cell.innerHTML = "red";
}
</script></head><body>
<table border="1"><tr>
 <td id="0" onclick="changeBackgroundBlue('0');">white</td>
 <td id="1" onclick="changeBackgroundRed(this);">white</td>
</tr></table></body></html>
```

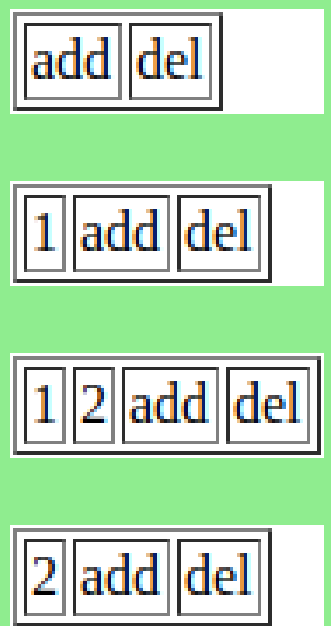


http://cgi.csc.liv.ac.uk/~ullrich/COMP519/examples/jsBG.html

Manipulating HTML elements (2)

It is not only possible to access HTML elements,
but also possible to add new ones and remove old ones on-the-fly

```
<html><head><title>Manipulating HTML elements (2)</title></head>
<body><table border="1"><tr>
<td onclick="addLeft(this);">add</td>
<td onclick="removeLeft(this.parentNode);">remove</td>
</tr></table><script>
function addLeft(node) {
  // Maintain a counter that is incremented with each call
  addLeft.counter = addLeft.counter || 0;
  addLeft.counter++;
  // Create a new TD element with counter as content
  newTD = document.createElement('td');
  newTD.innerHTML = addLeft.counter;
  newTD.setAttribute('id',addLeft.counter);
  // Add the new TD element before the current one
  node.parentNode.insertBefore(newTD,node);
}
function removeLeft(parent) {
  parent.removeChild(parent.firstChild);
}
</script></body></html>
```



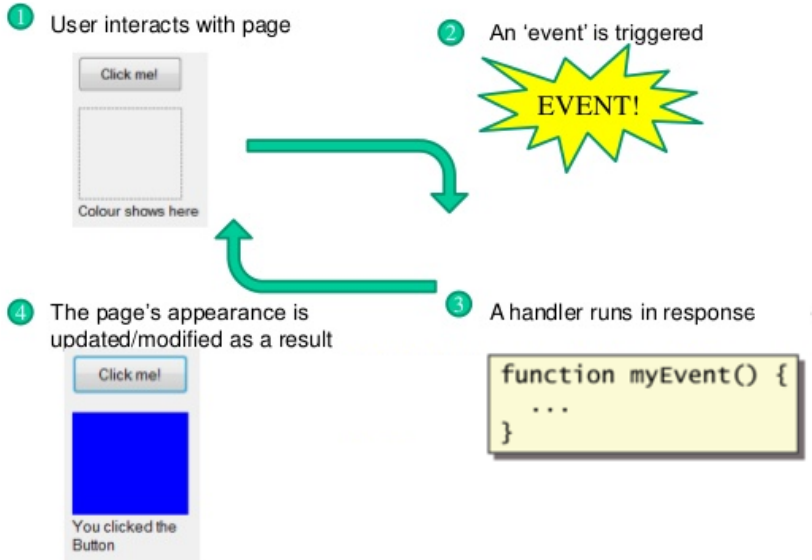
http://cgi.csc.liv.ac.uk/~ullrich/COMP519/examples/jsTD.html

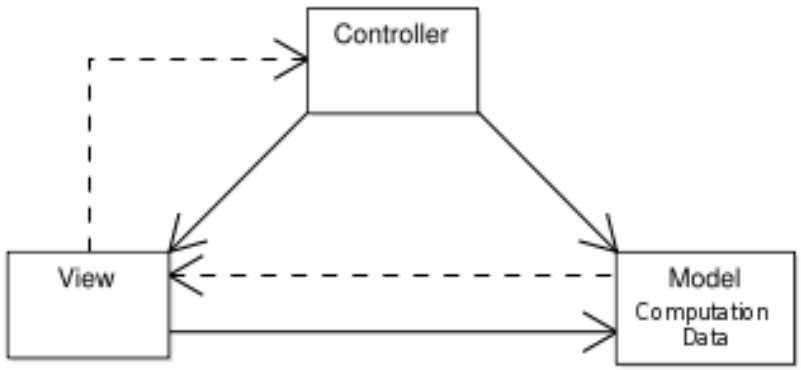
Event-driven JavaScript Programs

- The JavaScript programs we have seen so far were all executed sequentially
  - programs have a particular starting point
  - programs are executed step-by-step, involving control structures and function execution
  - programs reach a point at which their execution stops

Event-Driven JavaScript Programs

- Web applications are event-driven
  ⇝ they react to events such as mouse clicks and key strokes





nickywalters: What is Event Driven Programming?
SlideShare, 7 September 2014.
`https://tinyurl.com/ya58xbs9` [accessed 5/11/2017]

- With JavaScript,
  – we can define event handler functions for a wide variety of events
  – event handler functions can manipulate the `document` object (changing the web page in situ)

Event Handlers and HTML Elements

- HTML events are things, mostly user actions, that happen to HTML elements
- Event handlers are JavaScript functions that process events
- Event handlers must be associated with HTML elements for specific events
- This can be done via attributes

```
<input type="button" value="Help"  onclick="Help()">
```

- Alternatively, a JavaScript function can be used to add a handler to an HTML element

```
// All good browsers
window.addEventListener("load", Hello)
// MS IE browser
window.attachEvent("onload", Hello)
```

More than one event handler can be added this way to the same element for the same event or different events

Event Handlers and HTML Elements

- As our scripts should work with as many browsers as possible, we need to detect which method works:

```
if (window.addEventListener) {
   window.addEventListener("load", Hello)
} else {
   window.attachEvent("onload", Hello)
}
```

- Event handlers can also be removed

```
if (window.removeEventListener) {
   window.removeEventListener("load", Hello)
} else {
   window.detachEvent("onload", Hello)
}
```

Events: Load

- An (on)load event occurs when an object has been loaded
- Typically, event handlers for onload events are associated with the window object or the body element of an HTML document

```
<!DOCTYPE html>
<html lang="en-GB">
  <head>
    <title>Onload Example</title>
    <script>
      function Hello() { alert("Welcome to my page!")  }
    </script>
  </head>
  <body onload="Hello()">
    <p>Content of the web page</p>
  </body>
</html>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP519/examples/jsOnload.html

Events: Focus / Change

- A focus event occurs when a form field receives input focus by tabbing with the keyboard or clicking with the mouse
  ⇝ `onFocus` attribute
- A change event occurs when a select, text, or textarea field loses focus and its value has been modified
  ⇝ `onChange` attribute

Example:

```
<form name="form1" method="post" action="process.php">
  <select name="select" required
          onChange="document.form1.submit();">
    <option value="">Select a name</option>
    <option value="200812345">Tom Beck</option>
    <option value="200867890">Jim Kent</option>
  </select>
</form>
```

Events: Focus / Change

- A focus event occurs when a form field receives input focus by tabbing with the keyboard or clicking with the mouse
  ⇝ `onFocus` attribute
- A change event occurs when a select, text, or textarea field loses focus and its value has been modified
  ⇝ `onChange` attribute

```
<form>
<label>Temperature in Fahrenheit:</label>
<input type="text" id="fahrenheit" size="10" value="0"
 onchange="document.getElementById('celsius').value =
 FahrenheitToCelsius(parseFloat(
  document.getElementById('fahrenheit').value)).toFixed(1);"
 ><br>
<label>Temperature in Celsius:</label>
<input type="text" id="celsius"
 size="10" value="" onfocus="blur();"></form>
```

http://cgi.csc.liv.ac.uk/~ullrich/COMP519/examples/jsOnchange.html

Events: Blur / Click

- A blur event occurs when an HTML element loses focus
  $\rightsquigarrow$ `onBlur` attribute
- A click event occurs when an object on a form is clicked
  $\rightsquigarrow$ `onClick` attribute

Example:

```
<html><head><title>Onclick Example</title></head><body>
<form name="form1" action="">
  Enter a number here:
  <input type="text" size="12" id="number" value="3.1">
  <br><br>
  <input type="button" value="Double"
    onclick="document.getElementById('number').value =
    parseFloat(document.getElementById('number').value)
    * 2;">
</form></body></html>
```

`http://cgi.csc.liv.ac.uk/~ullrich/COMP284/examples/jsOnclick.html`

Events: MouseOver / Select / Submit

- A keydown event occurs when the user presses a key
  $\rightsquigarrow$ `onkeydown` attribute
- A mouseOver event occurs once each time the mouse pointer moves over an HTML element from outside that element
  $\rightsquigarrow$ `onMouseOver` attribute
- A select event occurs when a user selects some of the text within a text or textarea field
  $\rightsquigarrow$ `onSelect` attribute
- A submit event occurs when a user submits a form
  $\rightsquigarrow$ `onSubmit` attribute

Events and DOM

- When an event occurs, an event object is created
  ⇝ an event object has attributes and methods
  ⇝ event objects can be created by your code
  independent of an event occurring
- In most browsers, the event object is passed to event handler functions as an argument
- In most versions of Microsoft Internet Explorer, the most recent event can only be accessed via `window.event`

```
<html><body onKeyDown="processKey(event)">
 <script>
  function processKey(e) {
    e = e || window.event
    document.getElementById("message").innerHTML =
      String.fromCharCode(e.keyCode)+' has been pressed' }
 </script>
 <!-- key code will appear in the paragraph below -->
 <p id="message"></p>
</body></html>
```

Example: Two-Player Board Game

- We want to develop a two-player board game along the lines of Tic-Tac-Toe
- The full code is available at

  `http://cgi.csc.liv.ac.uk/~ullrich/COMP519/examples/jsBoard.html`

- The interface will consist of a 3x3 table representing the board and a section for messages, both of which will be generated dynamically

```
<body>
  <table id="t1"></table>
  <div id="m1"></div>
  <script>...</script>
</body>
```

Example: Two-Player Board Game

- Following the Model-View-Controller paradigm we need a model of the game, including the board and overall state of the

```
var board = [[0,0,0],[0,0,0],[0,0,0]];
var free  = 9;  // free positions on the board
var turn  = 1;  // alternates between 0 and 1
```

- We will use 0 to represent an empty position on the board
  1 to represent a position taken by player 1
  2 to represent a position taken by player 2
- We have a function that turns these values into 'nice' representations

```
function numToLetter(num) {
  switch (num) {
    case 0: return " "
    case 1: return "O"
    case 2: return "X"
  }
}
```

Example: Two-Player Board Game

- We need a function to show a message to the user and another to clear that message

```
function showMessage(message,style) {
   m1 = document.getElementById("m1");
   m1.innerHTML = message;
   m1.style.display = "block";
   m1.className = style;
}

function clearMessage() {
   m1 = document.getElementById("m1");
   m1.style.display = "none";
}
```

Example: Two-Player Board Game

- The `play` function implements the turn of a user

```
function play(x,y,event) {
  clearMessage();
  console.log("x = " + x + " y = " + y);
  console.log("b = " + board[y][x]);
  if (board[y][x] > 0) {
     showMessage("Grid position [" + x + "," + y +
                 "] already occupied","RedBG");
  } else {
    board[y][x] = 2 - turn;
    free--;
    event.target.innerHTML = numToLetter(2 - turn);
    turn = 1 - turn;
  }
}
```

- Arguments x and y are the co-ordinates on which the player as placed a piece
- `event` is the event that was triggered and `event.target` gives us the HTML element / table cell on which it was triggered

Example: Two-Player Board Game

- At the start we create a representation of the board

```
function init(table) {
  for (j=0; j<board.length; j++) {
    var tr = document.createElement("tr");
    table.appendChild(tr);
    for (i=0; i<board[j].length; x++) {
      var td  = document.createElement("td");
      var txt = document.createTextNode(
                    numToLetter(board[j][i]);
      td.appendChild(txt);
      td.setAttribute('id',"" + x + y);
      td.addEventListener("click",play.bind(null,i,j));
      tr.appendChild(td);
} } }
table = document.getElementById('t1');
init(table);
```

- `play.bind` makes sure that parameters x and y of `play` are bound to the current values of i and j

Example: Two-Player Board Game

- Finally, we add some CSS directives to improve the visual appearance of the game

```
<style>
td { border:          1px solid black;
     width:           2em;
     height:          2em;
     text-align:      center;
     vertical-align: middle;
 }
div.RedBG  {
     background-color: #f00;
}
div.GreenBG  {
     background-color: #0f0;
}
</style>
```

Example: Adding a Computer Player / Delays

```
var processing = false

async function play(x,y,event) {
  if (!processing) {
    processing = true;
    clearMessage();
    if (board[y][x] > 0) {
     showMessage("Grid position [" + x + "," + y +
                 "] already occupied","RedBG");
    } else {
      board[y][x] = 2 - turn; free--;
      event.target.innerHTML = numToLetter(2 - turn);
      turn = 1 - turn;
      await sleep(250); // sleep 250ms
      computerMove();
      processing = false
} } }

function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms)) }
```

Example: Two-Player Board Game

Possible improvements:

- We should detect that the board is full (`free == 0`) and end the game with an appropriate message
- We should detect a winning placement of pieces on the board, end the game and declare a winner
- If we have a computer player, then we need to implement `computerMove`

Revision and Further Reading

- Read
  - Chapter 21: Introduction to JavaScript: Events
  - Chapter 22: Using JavaScript: Meet the DOM

  of J. Niederst Robbins: Learning Web Design: A Beginner's Guide to HTML, CSS, JavaScript, and Web Graphics (5th ed). O'Reilly, 2018.
  E-book `https://library.liv.ac.uk/record=b5647021`
- Read
  - Chapter 10: The Document Object Model
  - Chapter 12: Events

  of N. C. Zakas: Professional JavaScript for Web developers. Wrox Press, 2009.
  Harold Cohen Library 518.59.Z21 or
  E-book `http://library.liv.ac.uk/record=b2238913`