5.1 JavaScript Execution Environment

- The JavaScript `Window` object represents the window in which the browser displays documents

- The `Window` object provides the largest enclosing referencing environment for scripts
  - All global variables are properties of `Window`

- 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 `Form` object has an `elements` array, which has references to the form's elements

5.2 The Document Object Model

- DOM 0 is supported by all JavaScript-enabled browsers (no written specification)

- DOM 1 was released in 1998
- DOM 2 was released in 2000
- DOM 3 is the latest approved standard (2004)

- The DOM is an abstract model that defines the interface between HTML documents and application programs—an API

- Documents in the DOM have a treelike structure

- 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"`

5.2 The Document Object Model

- IE9, FX3, and Chrome have ways to show the tree of a document

- See book for the necessary steps for both browsers to get the tree of a given document

```
<html xmlns = "http://www.w3.org/1999/xhtml">
  <head> <title> A simple table </title>
  </head>
  <body>
    <table border = "border">
      <tr>
        <th> </th>
        <th> Apple </th>
        <th> Orange </th>
      </tr>
      <tr>
        <th> Breakfast </th>
        <td> 0 </td>
        <td> 1 </td>
      </tr>
    </table>
  </body>
</html>
```

→SHOW Figure 5.1 (IE9)

→SHOW Figure 5.2 (FX3)

5.3 Element Access in JavaScript

- There are several ways to do it

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

```
<form action = "">
   <input type = "button"
          name = "pushMe" />
</form>
```

1. *DOM address*

```
document.forms[0].elements[0]
```

   Problem: document changes

2. *Element names* – requires the element and all of its ancestors (except `body`) to have `name` attributes

   - Example:

```
 <form name = "myForm"  action = "">
   <input type = "button"  name = "pushMe" />
</form>
```

```
   document.myForm.pushMe
```

5.3 Element Access in JavaScript (continued)

3. `getElementById` Method (defined in DOM 1)

- Example:

```
<form action = "">
   <input type = "button"  id = "pushMe" />
</form>

document.getElementById("pushMe")
```

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

```
 <form id = "topGroup">
   <input type = "checkbox"  name = "toppings"
          value = "olives" />
   ...
   <input type = "checkbox"  name = "toppings"
          value = "tomatoes" />
</form>
...
var numChecked = 0;
var dom = document.getElementById("topGroup");
for index = 0; index < dom.toppings.length;
    index++)
  if (dom.toppings[index].checked]
    numChecked++;
```

5.4 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*

Event	*Tag Attribute*
`blur`	`onblur`
`change`	`onchange`
`click`	`onclick`
`dblclick`	`ondblclick`
`focus`	`onfocus`
`keydown`	`onkeydown`
`keypress`	`onkeypress`
`keyup`	`onkeyup`
`load`	`onload`
`mousedown`	`onmousedown`
`mousemove`	`onmousemove`
`mouseout`	`onmouseout`
`mouseover`	`onmouseover`
`mouseup`	`onmouseup`
`reset`	`onreset`
`select`	`onselect`
`submit`	`onsubmit`
`unload`	`onunload`

5.4 Events and Event Handling (continued)

- 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 two ways:*

  1. When the user puts the mouse cursor over it and presses the left button

  2. When the user tabs to the element

- *Event handlers can be registered in two ways*:

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

```
onclick = "alert('Mouse click!');"
onclick = "myHandler();"
```

5.5 Handling Events from Body Elements

- Example: the `load` event - triggered when the loading of a document is completed

→SHOW `load.html`, `load.js`, & display

5.6 Handling Events from Button Elements

- Plain Buttons – use the `onclick` property

- *Radio buttons*

  - If the handler is registered in the markup, the particular button that was clicked can be sent to the handler as a parameter

  e.g., if `planeChoice` is the name of the handler and the value of a button is `172`, use

  ```
  onclick = "planeChoice(172)"
  ```

  - This is another way of choosing the clicked button

  → SHOW `radio_click.html`, `radio_click.js`, & display

5.6 Handling Events from Button Elements (continued)

2. (second way to register an event handler)
   - Assign the address of the handler function to the event property of the JavaScript object associated with the HTML element.

```
var dom = document.getElementById("myForm")
dom.elements[0].onclick = planeChoice;
```

- This registration must follow both the handler function and the XHTML form

- In this case, the `checked` property of a radio button object is used to determine whether a button is clicked
  - If the name of the buttons is `planeButton`

```
var dom = document.getElementById("myForm");
for (var index = 0;
  index < dom.planeButton.length; index++) {
  if (dom.planeButton[index].checked) {
    plane = dom.planeButton[index].value;
    break;
  }
}
```

→ SHOW `radio_click2.html` & `radio_click2.js`

5.6 Handling Events from Button Elements (continued)

- The disadvantage of specifying handlers by assigning them to event properties is that there is no way to use parameters

- The advantages of specifying handlers by assigning them to event properties are:

  1. It is good to keep HTML and JavaScript separate

  2. The handler could be changed during use

5.7 Handling Events from Textbox and Password Elements

- *The Focus Event*

  - Can be used to detect illicit changes to a text box by blurring the element every time the element acquires focus

  → SHOW `nochange.html` & `nochange.js`

5.7 Handling Events from Textbox and Password Elements (continued)

- *Checking Form Input*

- A good use of JavaScript, because it finds errors in form input before it is sent to the server for processing
  - So, it saves both:
    1. Server time, and

    2. Internet time

- *Things that must be done*:

  1. Detect the error and produce an `alert` message

  2. Inform the user of the error and present the correct format

5.7 Handling Events from Textbox and Password Elements (continued)

- To keep the form active after the event handler is finished, the handler must return `false`

- *Example* – comparing passwords

  - The form just has two password input boxes to get the passwords and Reset and Submit buttons

  - The event handler is triggered by the Submit button

5.7 Handling Events from Textbox and Password Elements (continued)

- *Handler actions*:

    1. If no password has been typed in the first box, focus on that box and return `false`

    2. If the two passwords are not the same, focus and select the first box and return `false` if they are the same, return `true`

→ SHOW `pswd_chk.html`, `pswd_chk.js`, `pswd_chkr.js`

- *Another Example* – Checking the format of a name and phone number

    - The event handler will be triggered by the `change` event of the text boxes for the name and phone number

    - If an error is found in either, an `alert` message is produced and both `focus` and `select` are called on the text box element

→ SHOW `validator.html`, `validator.js`, and `validatorr.js`

5.8 The DOM 2 Event Model

- Does not include DOM 0 features, but they are still supported by browsers

- DOM 2 is modularized—one module is `Events`, which has two submodules, `HTMLEvents` and `MouseEvents`, whose interfaces are `Event` (`blur`, `change`, etc.) and `MouseEvent` (`click`, `mouseup`, etc.)

- Event propagation

  - The node of the document tree where the event is created is called the *target node*

  - The *capturing phase* (the first phase)

    - Events begin at the root and move toward the target node

    - Registered and enabled event handlers at nodes along the way are run

  - The *second phase* is at the target node

    - If there are registered but not enabled handlers there for the event, they are run

  - The third phase is the *bubbling phase*
    - Event goes back to the root; all encountered registered but not enabled handlers are run

5.8 The DOM 2 Event Model (continued)

- Not all events bubble (e.g., `load` and `unload`)

- Any handler can stop further event propagation by calling the `stopPropagation` method of the `Event` object

- DOM 2 model uses the `Event` object method, `preventDefault`, to stop default operations, such as submission of a form, if an error has been detected

- Event handler registration is done with the `addEventListener` method

  - *Three parameters*:

    1. Name of the event, as a string literal

    2. The handler function

    3. A Boolean value that specifies whether the event is enabled during the capturing phase

```
node.addEventListener(
               "change", chkName, false);
```

5.8 The DOM 2 Event Model (continued)

- A temporary handler can be created by registering it and then unregistering it with `removeEventListener`

- The `currentTarget` property of `Event` always references the object on which the handler is being executed

- The `MouseEvent` interface (a subinterface of `Event`) has two properties, `clientX` and `clientY`, that have the x and y coordinates of the mouse cursor, relative to the upper left corner of the browser window

- An example: A revision of `validator`, using the DOM 2 event model

→ SHOW `validator2.html`, `validator2.js`, & `validator2r.js`

5.9 The `canvas` Element

- Creates a rectangle into which bit-mapped graphics can be drawn using JavaScript

- Optional attributes: `height`, `width`, and `id`

  - Default value for `height` and `width` are `150` and `300` pixels

  - The `id` attribute is required if something will be drawn

```
<canvas id = "myCanvas" height = "200"
        width = "400">
   Your browser does not support the canvas
     element
</canvas>
```

5.10 The `navigator` object

- Indicates which browser is being used

- Two useful properties

  1. The `appName` property has the browser's name

  2. The `appVersion` property has the version #

- Microsoft has chosen to set the `appVersion` of IE9 to 5 (?)

- Firefox has chosen to set the `appVersion` of FX3 to 5.0 (?) and the name to Netscape (?)

→SHOW `navigate.html` & `navigate.js`

5.11 DOM Tree Traversal and Modification

- *Traversal properties*: `parentNode`, `previousSibling`, `nextSibling`, `firstChild`, `childNodes`, and `lastChild`

- For example, if there is an unordered list with the id `myList`, the number of list items in the list can be displayed with:

```
var dom = document.getElementById("myList");
var listItems = dom.childNodes.length;
document.write("Number of list items is: " +
               listItems + "<br />");
```

- *Modification methods*: `insertBefore`, `replaceChild`, `removeChild`, `appendChild`