Web Engineering SE371

Skander Turki, Prince Sultan University

Chapter 5

Using JavaScript in the front-end

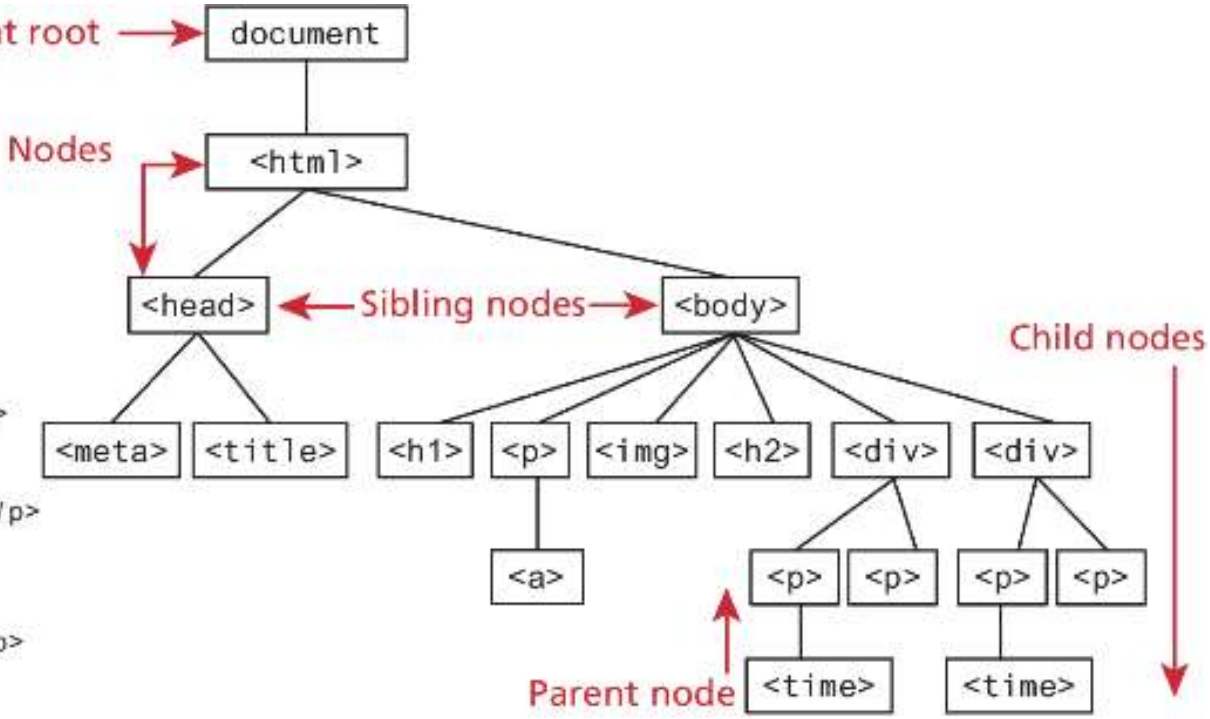
JavaScript Output (third-party libraries): sweetalert

```
<script src="https://cdn.jsdelivr.net/npm/sweetalert2@11">
</script>
.....
```


```
Swal.fire({
  icon: "error",
  title: "Something went wrong...",
  text: "Your data could not be saved!",
  footer: '<a href="/issues">Why do I have this issue?</a>'
});
```

The Document Object Model (DOM)

```
<html>
<head>
   <meta charset="utf-8">
   <title>Share Your Travels</title>
</head>
<body>
   <h1>Share Your Travels</h1>
   <p>Photo of Conservatory Pond in
      <a href="#">Central Park</a>
   </p>
   <img src="park.jpg" alt="Central Park" />
   <h2>Reviews</h2>
   <div id="latestComment">
      <p>Ricardo on <time>2021-05-23</time></p>
      <p>Easy on the HDR buddy.</p>
   </div>
   <div>
      <p>Susan on <time>2021-11-18</time></p>
      <p>I love Central Park.</p>
   </div>
</body>
</html>
```



Document Object

The **DOM document object** is the root JavaScript object representing the entire HTML document. It is globally accessible via the **document** object reference.

The properties of a document cover information about the page. Some are read-only, but others are modifiable. Like any JavaScript object, you can access its properties using either dot notation or square bracket notation

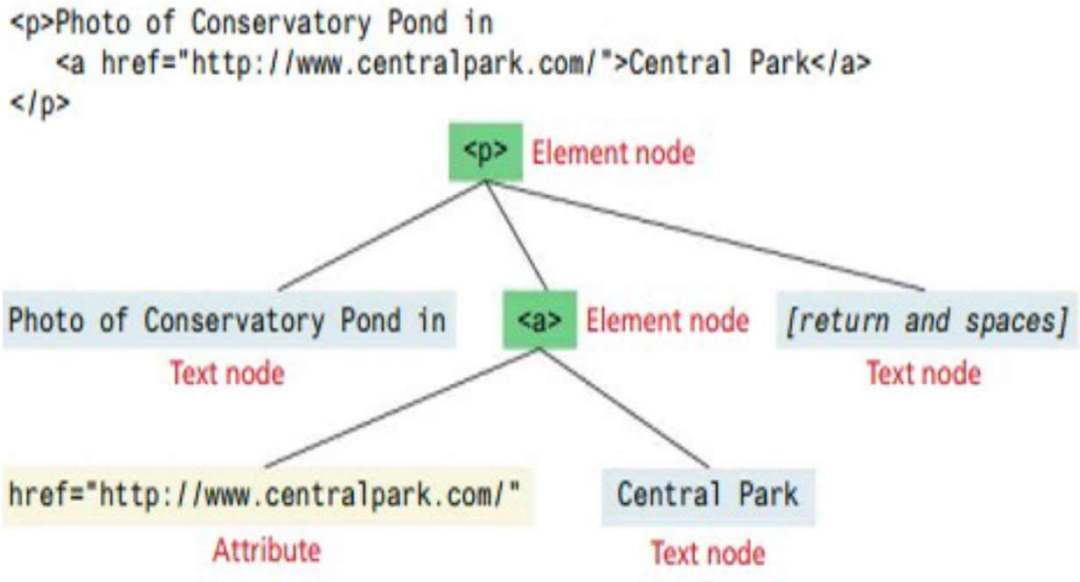
```
// retrieve the URL of the current page
let a = document.URL;
// retrieve the page encoding, for example ISO-8859-1
let b = document["inputEncoding"];          // equivalent to document.inputEncoding
```

DOM Nodes and NodeLists

In the DOM, each element within the HTML document is called a **node**.

The DOM also defines a specialized object called a **NodeList** that represents a collection of nodes. It operates very similarly to an array.

Many programming tasks that we typically perform in JavaScript involve finding one or more nodes and then modifying them.



Some Essential Node Object Properties

- **childNodes** A NodeList of child nodes for this node

```
> document.getRootNode().childNodes
<· ▸NodeList(2) [<!DOCTYPE html>, html]
```

- **firstChild** First child node of this node
- **lastChild** Last child of this node
- **nextSibling** Next sibling node for this node
- **nodeName** Name of the node
- **parentNode** Parent node for this node
- **previousSibling** Previous sibling node for this node
- **textContent** Represents the text content (stripped of any tags) of the node

document object Methods: Selection

They allow you to select one or more document elements. The oldest 3 are: **getElementById("*id*"), getElementsByClassName("*name*")** and **getElementsByTagName("*name*")**

```
let node = document.getElementById("latest");
```

```
<body>
   <h1>Reviews</h1>
   <div id="latest">
      <p>By Ricardo on <time>2016-05-23</time</p>
      <p class="comment">Easy on the HDR buddy.</p>
   </div>
   <hr/>
   <div>
      <p>By Susan on <time>2016-11-18</time></p>
      <p class="comment">I love Central Park.</p>
   </div>
   <hr/>
</body>
```

```
let list2 = document.getElementsByClassName("comment");
```

```
let list1 = document.getElementsByTagName("div");
```

document object Methods: Query Selection

The newer **querySelector**() and **querySelectorAll**() methods allow you to query for DOM elements the same way you specify CSS styles

```
querySelectorAll("nav ul a:link")

<body>
    <nav>
        <ul>
            <li><a href="#">Canada</a></li>
            <li><a href="#">Germany</a></li>
            <li><a href="#">United States</a></li>
        </ul>
    </nav>
    <div id="main">
        Comments as of
        <time>November 15, 2012</time>
        <div>
            <p>By Ricardo on <time>September 15, 2012</time></p>
            <p>Easy on the HDR buddy.</p>
        </div>

        <div>
            <p>By Susan on <time>October 1, 2012</time></p>
            <p>I love Central Park.</p>
        </div>
    </div>
    <footer>
        <ul>
            <li><a href="#">Home</a> | </li>
            <li><a href="#">Browse</a> | </li>
        </ul>
    </footer>
</body>

querySelectorAll("#main div time")
querySelector("#main>time")
querySelector("footer")
```

Element Node Object

Element Node object represents an HTML element in the hierarchy, contained between the opening <> and closing </>.

Every Element Node has also these properties:

- **classList** A read-only list of CSS classes assigned to this element. This list has a variety of helper methods for manipulating this list.
- **className** The current value for the class attribute of this HTML element.
- **id** The current value for the id of this element.
- **innerHTML** Represents all the content (text and tags) of the element.
- **style** The style attribute of an element. This returns a CSSStyleDeclaration object that contains sub-properties that correspond to the various CSS properties.
- **tagName** The tag name for the element.

Extra Properties for Certain Tag Types

Property	Description	Tags
href	Used in <a> tags to specify the linking URL.	a
name	Used to identify a tag. Unlike id which is available to all tags, name is limited to certain form-related tags.	a, input, textarea, form
src	Links to an external URL that should be loaded into the page (as opposed to href which is a link to follow when clicked).	img, input, iframe, script
value	Provides access to the value attribute of input tags. Typically used to access the user's input into a form field.	input, textarea, submit

Accessing elements and their properties

```
<p id="here">hello <span>there</span></p>
<ul>
    <li>France</li>
    <li>Spain</li>
    <li>Thailand</li>
</ul>
<div id="main">
    <a href="somewhere.html">
        <img src="whatever.gif" class="thumb">
    </a>
</div>

<script>

const node = document.getElementById("here");
console.log(node.innerHTML); // hello <span>there</span>
console.log(node.textContent); //“hello there”

const items = document.getElementsByTagName("li");
for (let i=0; i<items.length; i++) {
    // outputs: France, then Spain, then Thailand
    console.log(items[i].textContent);
}

const link = document.querySelector("#main a");

console.log(link.href); // outputs: somewhere.html

const img = document.querySelector("#main img");
console.log(img.src); // outputs: whatever.gif

console.log(img.className); // outputs: thumb

</script>
```

LISTING 9.1 Accessing elements and their properties

Pearson

Source code: chapter04/08_dom

Changing an Element's Style

To programmatically modify the styles associated with a particular element one must change the properties of the style property for that element

For instance, to change an element's background color and add a three pixel border, we could use the following code:

```
const node = document.getElementById("someId");
node.style.backgroundColor = "#FFFF00";
node.style.borderWidth = "3px";
```

How CSS styles can be programmatically manipulated in JavaScript

While you can directly change CSS style elements via this **style** property, it is preferable to change the appearance of an element using the **className** or **classList** properties

```
<style>
    .box {
        margin: 2em; padding: 0;
        border: solid 1pt black;
    }
   .yellowish { background-color: #EFE63F; }
   .hide { display: none; }
</style>
<main>
   <div class="box">
    ...
   </div>
</main>
```

```
var node = document.querySelector("main div");

❶ node.className = "yellowish";
   This replaces the existing class specification with this one. Thus the <div> no longer has the box class

❷ node.classList.remove("yellowish");
  node.classList.add("box");
   Removes the specified class specification and adds the box class

❸ node.classList.add("yellowish");
   Adds a new class to the existing class specification

❹ node.classList.toggle("hide");
   If it isn't in the class specification, then add it

❺ node.classList.toggle("hide");
   If it is in the class specification, then remove it
```

```
Equivalent to:

❶ <div class="yellowish">

❷ <div class="">
  <div class="box">

❸ <div class="box yellowish">

❹ <div class="box yellowish hide">

❺ <div class="box yellowish">
```

InnerHTML vs textContent vs DOM Manipulation

You can programmatically access the content of an element node through its **innerHTML** or **textContent** property. These properties can also be used to modify the content of any given element.

For instance, you could change the content of the using the following:

```
const div = document.querySelector("#main");
div.innerHTML = '<a href="#"><img src="ab.gif"></a>';
```

This replaces the existing content with the new content. **But, every time innerHTML is set, the HTML has to be parsed, a DOM constructed, and inserted into the document. This takes time.**

Exercise 2 (innerHTML + functions)

1- Write a function expression ***makeArticle*** that produces an HTML code that represents an article containing an h2 element and three p elements as follows:

Example:

makeArticle (“manager”, “Director”, “Salah”, “Abed”, salahabed@abc.com);

This call to the function should produce the HTML code displayed on the right inside into the node with the given id.

2- Rewrite this function as an arrow function

3- Create a constructor function for the Employee entity, add a function to its properties that returns the given HTML.

```
<div id=“manager”></div>
```

```
<div id=“manger”>
  <article>
    <h2>Position: Director</h2>
    <p>Name: Salah</p>
    <p>Last Name: Abed</p>
    <p>Email: salahabed@abc.com</p>
  </article>
</div>
```

Source code: chapter04/09_ex_innerhtml

DOM family relations

Each node in the DOM has a variety of “family relations” properties and methods for navigating between elements and for adding or removing elements from the document hierarchy.

Child and sibling properties can be an unreliable mechanism for selecting nodes and thus, in general, you will instead use selector methods

DOM Manipulation Methods

- **appendChild** Adds a new child node to the end of the current node.
- **createElement** Creates an HTML element node.
- **createTextNode** Creates a text node.
- **insertAdjacentElement** Inserts a new child node at one of four positions relative to the current node.
- **insertAdjacentText** Inserts a new text node at one of four positions relative to the current node.
- **removeChild** Removes a child from the current node.
- **replaceChild** Replaces a child node with a different child.

```
<!-- beforebegin -->
<p>
<!-- afterbegin -->
foo
<!-- beforeend -->
</p>
<!-- afterend -->
```

Visualizing the DOM modification

We want to add a new <p> to this :

```
<div id="first">
    <h1>DOM Example</h1>
    <p>Existing element</p>
</div>
```

Visualizing the DOM elements

```
<div>
    <h1> "DOM Example" </h1>
    <p> "Existing element" </p>
</div>
```

1. Create a new text node

```
"this is dynamic"
```

```
const text = document.createTextNode("this is dynamic");
```

2. Create a new empty <p> element

```
<p></p>
```

```
const p = document.createElement("p");
```

Visualizing the DOM modification (ii)

3 Add the text node to new element

```
p.appendChild(text);
```

 "this is dynamic"

4 Add the element to the

```
const first = document.getElementById("first");
first.appendChild(p);
```

```
<div id="first">
    <h1>DOM Example</h1>
    <p>Existing element</p>
    <p>this is dynamic</p>
</div>
```

```
<div>
    <h1> "DOM Example" </h1>
    <p> "Existing element" </p>
    <p> "this is dynamic" </p>
</div>
```

Same Exercise p65 (DOM manipulation)

Modify makeArticle to use DOM manipulation functions (createElement, appendChild, etc).

```
<article>

  <h2>Position: Director</h2>

  <p>Name: Salah</p>

  <p>Last Name: Abed</p>

  <p>Email: salahabed@abc.com</p>

</article>
```

```
4   const makeArticle = function (displayElement, position, name, lastName, email) {
5     let p = document.getElementById(displayElement);
6     let art = document.createElement("article");
7     let h2 = document.createElement("h2");
8     h2.appendChild( document.createTextNode('Position: ' + position) );
9     let p1 = document.createElement("p");
10    p1.appendChild( document.createTextNode('Name: ' + name) );
11    let p2 = document.createElement("p");
12    p2.appendChild( document.createTextNode('Last Name: ' + lastName ) );
13    let p3 = document.createElement("p");
14    p3.appendChild( document.createTextNode('Email: ' + email) );
15
16    art.appendChild(h2);
17    art.appendChild(p1);
18    art.appendChild(p2);
19    art.appendChild(p3);
20    p.appendChild(art);
21  }
22
23  function getvalue(id) {
24    return document.getElementById(id).value;
25  }
26
27  function clearDisplay(displayElement){
28    let p = document.getElementById(displayElement);
29    p.removeChild(p.firstElementChild);
30  }
```

Using the Dataset Property

- You can use the **dataset** property of DOM elements to store data, which provides read/write access to custom data attributes (data-*).

```
<div id="container" data-userid="2356" data-user-name="Salah">
  <p id="display">The user ID is:</p>
</div>

<script>
  const container = document.getElementById("container");
  const user_id = container.dataset.userid;
  const user_name = container.dataset.userName;

  container.appendChild(document.createTextNode(`User info: ${user_id}, ${user_name}`));
</script>
```

Source code: chapter04/14_datasets

Handling Events

DOM Timing

Before finishing this section on using the DOM, it should be emphasized that the timing of any DOM code is very important.

You cannot access or modify the DOM until it has been loaded.

If the DOM programming is written *after* the markup that *should* ensure that the elements exist in the DOM before the code executes.

To wait until we know for sure that the DOM has been loaded requires knowledge from our next section on **event handling**.

JavaScript Event Handling

Implementing an Event Handler

An event handler is first defined, then registered to an element node object.

Registering an event handler requires passing a callback function to the **addEventListener()**

```
<input type="submit" id="btn">
<script>
  function simpleHandler() {
     alert("button was clicked");
  }
  const btn = document.querySelector("#btn");
  btn.addEventListener("click", simpleHandler);
</script>
```

1 Event handler defined

2 Event handler registered

Handling events with anonymous functions

It is much more common to make use of an *anonymous function* passed to **addEventListener**()

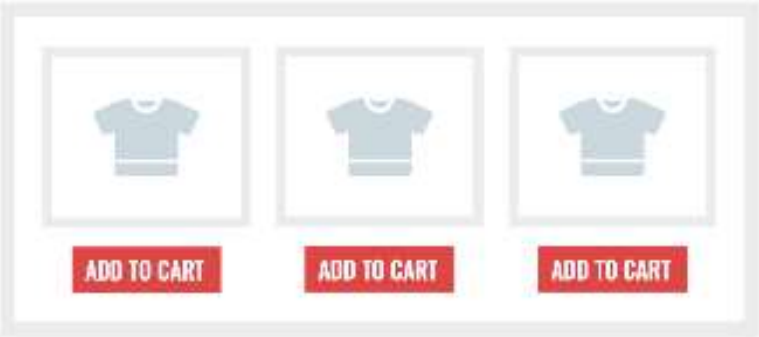
```
const btn = document.getElementById("btn");
btn.addEventListener("click", function () {
    alert("used an anonymous function");
});

document.querySelector("#btn").addEventListener("click", function (){
    alert("a different approach but same result");
});

document.querySelector("#btn").addEventListener("click", () => {
    alert("arrow syntax but same result");
});
```

LISTING 9.3 Listening to an event with an anonymous function, three versions

Event handling with NodeList arrays




```
<ul id="list">
  <li>
    <img src="a.png" ... >
    <button>Add To Cart</button>
  </li>
  <li>
    <img src="b.png" ... >
    <button>Add To Cart</button>
  </li>
  <li>
    <img src="c.png" ... >
    <button>Add To Cart</button>
  </li>
</ul>
```

```
// select all the buttons
const btns = document.querySelectorAll("#list button");

// this won't work and will generate error
btns.addEventListener("click", function () { ... });
```

```
// instead must loop through node list ...
for (let bt of btns) {
   // ...and assign event listener to each node
   bt.addEventListener("click", function () { ... });
}
```

Remember that a node list (i.e., array of nodes) doesn't support event listeners. Only individual node objects have the `addEventListener()` method defined.

Source code: chapter04/12_events_window

Page Loading and the DOM

To ensure your DOM manipulation code executes *after* the page is loaded, use one of the following two different page load events.

- **window.load** Fires when the entire page is loaded. This includes images and stylesheets, so on a slow connection or a page with a lot of images, the load event can take a long time to fire.
- **document.DOMContentLoaded** Fires when the HTML document has been completely downloaded and parsed. Generally, this is the event you want to use.

Using one of these, your DOM coding can now appear anywhere, including within the **<head>** element as long as you do not try to access the DOM.

Source code: chapter04/13_events_DOM

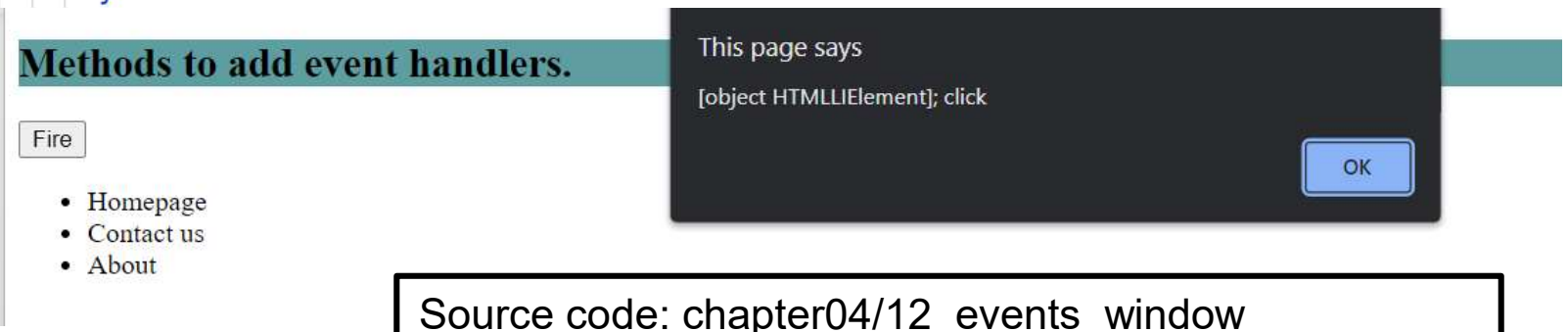
Wrapping DOM code within a DOMContentLoaded event handler

```
document.addEventListener('DOMContentLoaded', function() {
  const menu = document.querySelectorAll("#menu li");
  for (let item of menu) {
    item.addEventListener("click", function () {
      item.classList.toggle('shadow’);
    });
  }
  const heading = document.querySelector("h3");
  heading.addEventListener('click', function() {
    heading.classList.toggle('shadow’);
  });
});
```

Event Object

- When an event is triggered, the browser will construct an **event object** that contains information about the event.
- Your event handlers can access this event object simply by including it as an argument to the callback function (this event object parameter is often named *e*)

```
const heading = document.querySelector("h2");
heading.addEventListener('click', function(e) {
                                    heading.classList.toggle('shadow');
                                    alert(e.target + "; " + e.type);
                                  }
                                );
```



Source code: chapter04/12_events_window

Event Object Example


```
<ul id="menu">
  <li>Home</li>
  <li>About</li>
  <li>Products</li>
  <li>Contact</li>
</ul>
```

```
const menu = document.querySelectorAll("#menu li");
for (let item of menu) {
  item.addEventListener("click", menuHandler );
}
function menuHandler(e) {
    const x = e.clientX;
    const y = e.clientY;
    displayArrow(x,y);
    e.target.classList.toggle("selected");
    performMenuAction(e.target.innerHTML);
}
```

By receiving the event object as a parameter and using it to reference the clicked item, the `menuHandler()` function will work no matter where it is located.

Click events include the on-screen pixel location of the mouse cursor.

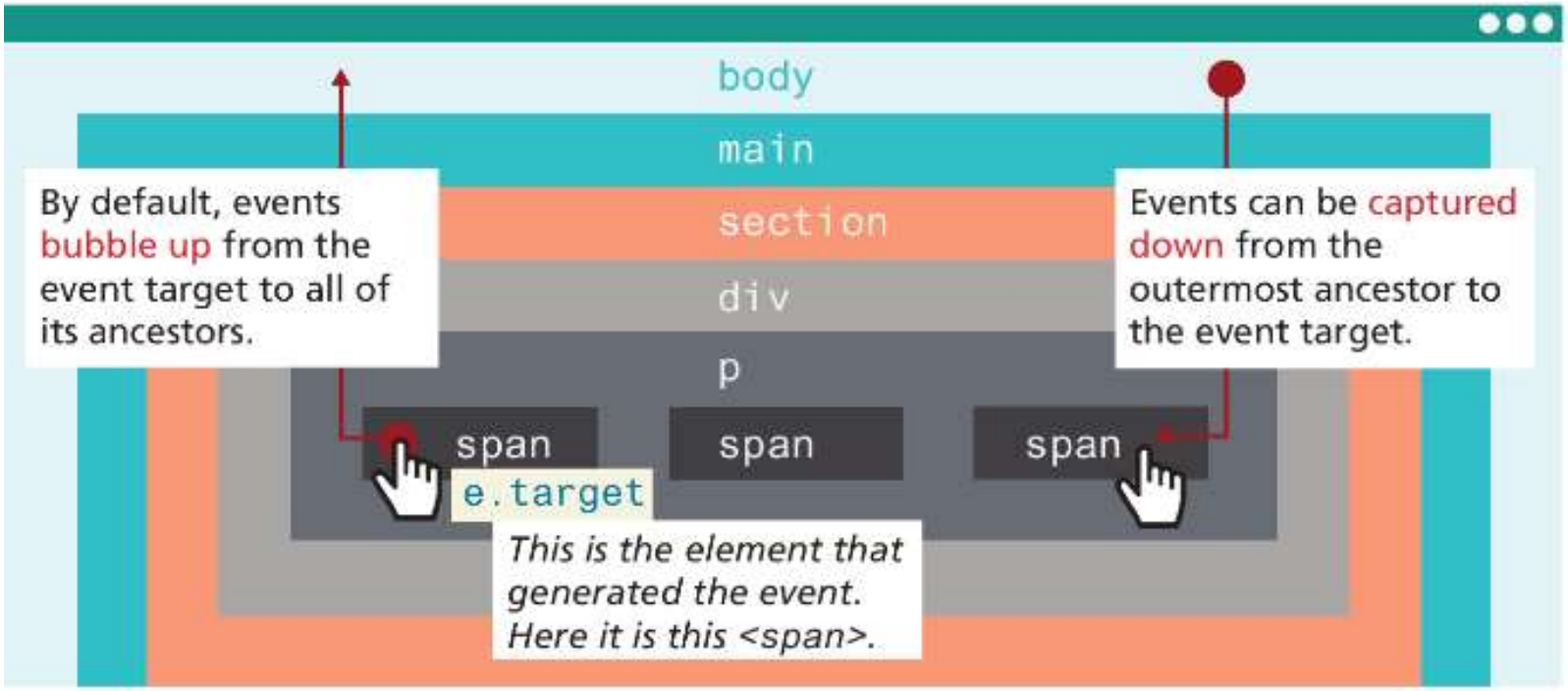
The `e.target` object in this case is referencing the clicked `<li>` item.

Event Propagation

When an event fires on an element that has ancestor elements, the event propagates to those ancestors. There are two distinct phases of propagation:

- In the **event capturing phase**, the browser checks the outermost ancestor (the <html> element) to see if that element has an event handler registered for the triggered event, and if so, it is executed (if configured to fire at this phase, *see code example_18_event_propagation*). It then proceeds to the next ancestor and performs the same steps; this continues until it reaches the element that triggered the event (that is, the **event target**).
- In the **event bubbling phase**, the opposite occurs. The browser checks if the element that triggered the event has an event handler registered for that event, and if so, it is executed.

Event capture and bubbling



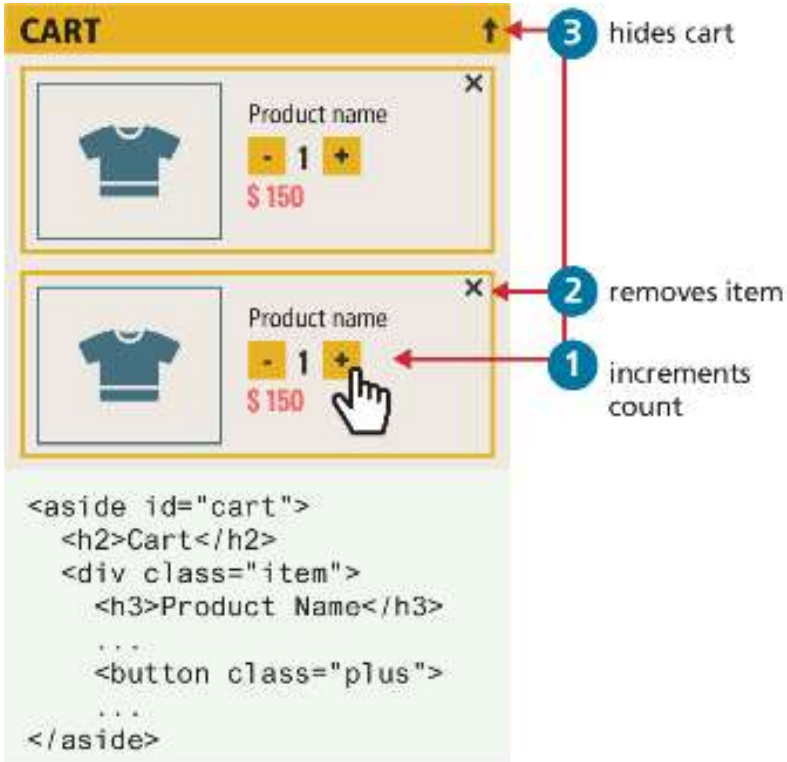
`e.currentTarget` *This is the element whose event handler is currently being executed. In this example, it could the or any of its ancestors.*

Problems with event propagation

Occasionally, the bubbling of events can cause problems. For instance consider elements nested within one another, each with its own on-click behaviors.

When the user clicks on the increment count button, the click handler for the increment <button> will trigger first. Unfortunately, it will then trigger the click event for the , and the <aside> element!

Thankfully, there is a solution to such problems. The **stopPropagation()** method of the event argument object will stop event propagation.



Stopping event propagation

```
const btns = document.querySelectorAll(".plus");
for (let b of btns) {
   b.addEventListener("click", function (e) {
      e.stopPropagation();
      incrementCount(e);
   });
}

const items = document.querySelectorAll(".item");
for (let it of items) {
   it.addEventListener("click", function (e) {
      e.stopPropagation();
      removeItemFromCart(e);
   });
}
```

```
const aside = document.querySelector("aside#cart");
aside.addEventListener("click", function () {
      minimizeCart();
});
```

LISTING 9.5 Stopping event propagation

Source code: chapter04/18_event_propagation

Event Delegation

To avoid creating duplicate event handlers for each element within a **NodeList**, an alternative is to use **event delegation** where we assign a single listener to the parent and make use of event bubbling

Suppose we have numerous image thumbnails within a parent element, similar to the following:

```
<body>
  <header>...</header>
  <main>
    <section id="list">
      <h2>Section Title</h2>
        <img ... />
        <img ... />
        ...
    </section>
  </main>
</body>
```

Event Delegation (ii)

Now what if you wanted to do something special when the user clicks the mouse on an <img>

You would probably write something like the following:

Notice that this solution adds an event listener to every <img> element.

```
const images =   document.querySelectorAll("#list img");
for (let img of images) {
        img.addEventListener("click", someHandler);
}
```

Event Delegation (iii)

Instead, we can add a single listener to the parent element, as shown in the following code

Since the user can click on any element within the <section> element, the click event handler needs to determine if the user has clicked on one of the <img> elements within it.

```
const parent = document.querySelector("#list");
parent.addEventListener("click", function (e) {
   // e.target is the object that generated the event.
   // to verify that e.target exists and that it is one of the
   // <img> elements. Note: NodeName always returns
   //upper case
   if (e.target && e.target.nodeName == "IMG") {
      doSomething(e.target);
    }
});
```

Event Types

There are many different types of events that can be triggered in the browser. Perhaps the most obvious event is the click event, but JavaScript and the DOM support several others.

- mouse events,
- keyboard events,
- touch events,
- form events, and
- frame events.

Mouse Events

- **click** The mouse was clicked on an element.
- **dblclick** The mouse was double clicked on an element.
- **mousedown** The mouse pressed down over an element.
- **mouseup** The mouse was released over an element.
- **mouseover** The mouse was moved (not clicked) over an element.
- **mouseout** The mouse was moved off of an element.
- **mousemove** The mouse was moved while over an element.

Tiles Game

\	A	B	C	D	E	F
1						
2						
3						
4						
5						
6						

Practice: Try to build a board game where the tiles are colored when a mouse event occurs.

Source code: chapter04/15_tiles_game

Keyboard Events

- **keydown** The user is pressing a key (this happens first).
- **keyup** The user releases a key that was down (this happens last).

```
document.getElementById("pagebody").addEventListener("keydown", function (e) {
   // get the raw key code
   let keyPressed=e.key;
   // convert to string
   let character=String.fromCharCode(keyPressed);
   alert("Key " + character + " was pressed");
});
```

Form Events

- **blur** Triggered when a form element has lost focus (i.e., control has moved to a different element), perhaps due to a click or Tab key press.
- **change** Some <input>, <textarea>, or <select> field had their value changed. This could mean the user typed something or selected a new choice.
- **focus** Complementing the blur event, this is triggered when an element gets focus (the user clicks in the field or tabs to it).
- **reset** HTML forms have the ability to be reset. This event is triggered when that happens.
- **select** When the user selects some text. This is often used to try and prevent copy/paste.
- **submit** When the form is submitted this event is triggered. We can do some prevalidation of the form in JavaScript before sending the data on to the server.

Handling the submit event

```
document.querySelector("#loginForm").addEventListener("submit",
function(e) {
        let pass = document.querySelector("#pw").value;
        if (pass == "") {
                alert ("enter a password");
                e.preventDefault();     // prevents form submission
        }
});
```

LISTING 9.8 Handling the submit event

Source code: chapter04/16_form_submit_event

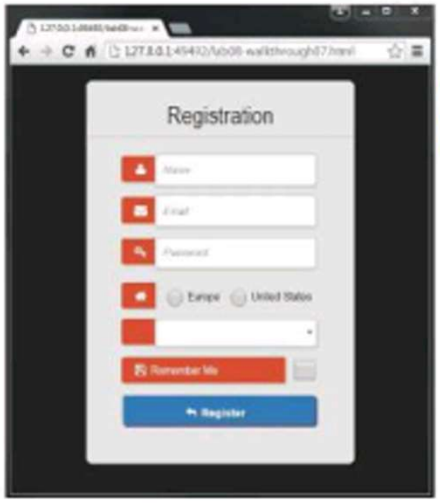
Forms in JavaScript

JavaScript within forms is more than just the client-side validation of form data; JavaScript is also used to improve the user experience of the typical browser-based form.

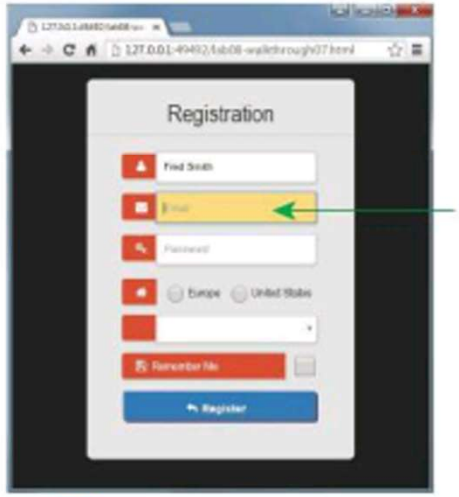
As a result, when working with forms in JavaScript, we are typically interested in three types of events:

- movement between elements,
- data being changed within a form element, and
- the final submission of the form.

Responding to form movement events



How form appears when no controls have the focus



When a control has the focus, then change its background color

```
// This function is going to get called every time the focus or blur events are
// triggered in one of our form's input elements.
function setBackground(e) {
   if (e.type == "focus") {
      e.target.style.backgroundColor = "#FFE393";
   }
   else if (e.type == "blur") {
      e.target.style.backgroundColor = "white";
   }
}

// set up the event listeners only after the DOM is loaded
window.addEventListener("load", function() {
   const cssSelector = "input[type=text],input[type=password]";
   const fields = document.querySelectorAll(cssSelector);
   for (let f of fields) {
      f.addEventListener("focus", setBackground);
      f.addEventListener("blur", setBackground);
   }
});
```

Here we use the `style` property instead of the `classList` property because of specificity conflicts (that is, attribute selectors override class selectors).

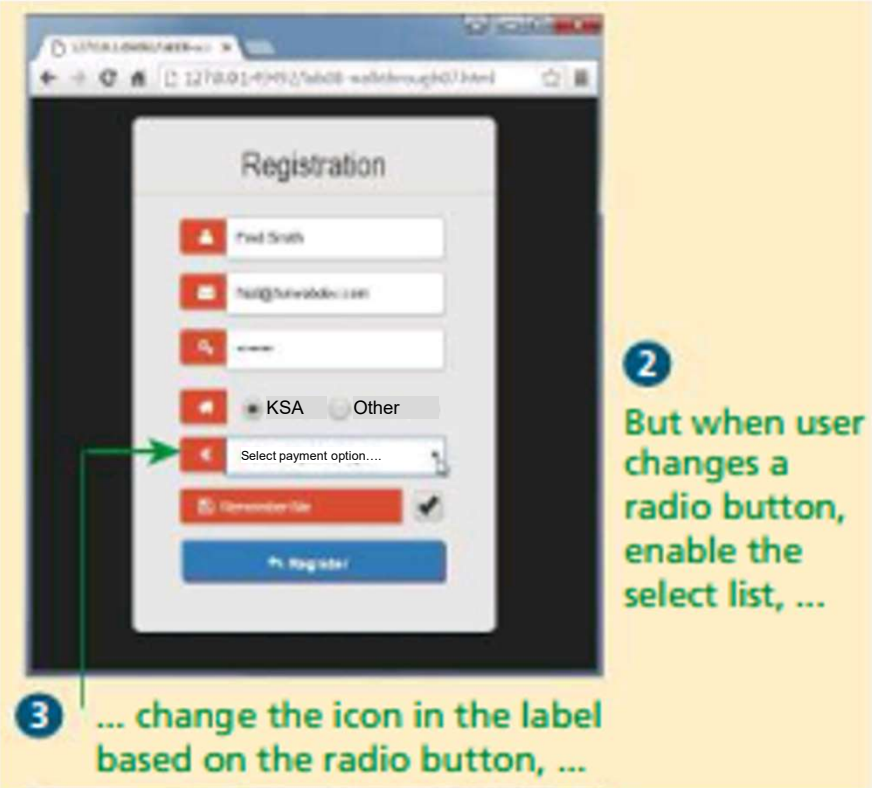
Selects the fields that will change.

Assigns the `setBackground()` function to change the background color of the control depending upon whether it has the focus.

Responding to Form Changes Events

We may want to change the options available within a form based on earlier user entry. For instance, we may want the payment options to be different based on the value of the region radio button.

We can add event listeners to the change event of the radio buttons; when one of these buttons changes its value, then the callback function will set the available payment options based on the selected region.



Validating a Submitted Form

Form validation continues to be one of the most common applications of JavaScript.

Checking user inputs to ensure that they follow expected rules must happen on the server side for security reasons (in case JavaScript was circumvented); checking those same inputs on the client side using JavaScript will reduce server load and increase the perceived speed and responsiveness of the form.

Some of the more common validation activities include email validation, number validation, and data validation.

In practice, regular expressions are used to concisely implement many of these validation checks.

Empty Field Validation

```
const form = document.querySelector("#loginForm");
form.addEventListener("submit", (e) => {
  const fieldValue = document.querySelector("#username").value;
  if (fieldValue == null || fieldValue == "") {
    // the field was empty. Stop form submission
    e.preventDefault();
    // Now tell the user something went wrong
    console.log("you must enter a username");
  }
});
```

LISTING 9.10 A simple validation script to check for empty fields

Determining which items in multiselect list are selected

```
const multi = document.querySelector("#listbox");
// using the options technique loops through each option and check if it is selected
for (let i=0; i < multi.options.length; i++) {
   if (multi.options[i].selected) {
      // this option was selected, do something with it ...
      console.log(multi.options[i].textContent);
   }
}

// the selectedOptions technique is simpler ... it only loops through the selected options
for (let i=0; i < multi.selectedOptions.length; i++) {
   console.log(multi.selectedOptions[i].textContent);
}
```

LISTING 9.11 Determining which items in multiselect list are selected

Source code: chapter04/20_multiselect_validation

Javascript validation examples

Some browsers may not support HTML5 validation, in addition to the fact that we want to have more control over how we react to bad user input, it is always better to use javascript validation:

```
<form ….. >
<input id="textA" type="number" max="100">
<button onclick="validateTextA()">Submit
</button>

<p id="output"></p>

</form>
```

```
<script>
  function validateTextA() {
    var value = "";
    if(document.getElementById("textA")
        .validity.rangeOverflow) {
      value = "The value must not be greater than 100";
    }
    document.getElementById("output")
        .innerHTML = value;
  }
</script>
```

Number Validation

Unfortunately, no simple functions exist for number validation. Using parseFloat(), isNAN(), and isFinite(), you can write your own number validation function.

Validating email, phone numbers, or social security numbers would include checking for blank fields and making use of isNumeric and/or regular expressions.

```
function isNumeric(n) {
        return !isNaN(parseFloat(n)) && isFinite(n);
}
// you have to use both because 1/0 is considered a number
```

Submitting Forms using JS

To submit a form using JavaScript simply call the **submit()** method:

```
const formExample = document.getElementById("loginForm");
formExample.submit();
```

This is often done in conjunction with calling preventDefault() on the submit event.

Javascript validation examples (2)

Validating that two entered emails are the same: We can use ***onchange*** *as it* is triggered when we type enter or leave the field, so this is the right event handler to use.

```
<form>
  <label>Preferred email address:
  <input type="email" id="email_addr"
name="email1"
placeHolder="user@provider.domain"
required></label>

  <label>Repeat email address:
  <input type="email" id="email_repeat"
name="email2" required
onchange="check()"></label>

  <input type = 'submit' value = "Send">
</form>
```

```
<script>
  function check(e) {
    var email1 =
        document.getElementById('email_addr');
    var email2 =
        document.getElementById('email_repeat');
    if ( email1.value !== email2.value) {
            e.preventDefault(); // no need (onchange)
            alert("The two emails have to match");
    }
  }
</script>
```

Javascript validation examples (3)

The problem with the previous code is that **we still can submit the form**. So, we need to add a second check that blocks the submission of the Http request:

```
<form>
  <label>Preferred email address:
  <input type="email" id="email_addr"
name="email1"
placeholder="user@provider.domain"
required></label>

  <label>Repeat email address:
  <input type="email" id="email_repeat"
name="email2" required onchange="return
check()"></label>

  <input type = 'submit' value ="Send"
                    onclick="check(this);">
</form>
```

```
<script type="text/javascript">
  function check(e) {
    var email1 =
        document.getElementById('email_addr');
    var email2 =
        document.getElementById('email_repeat');
    if ( email1.value !== email2.value) {
       e.preventDefault();
       alert("The two emails have to match");
    }
  }
</script>
```

Regular Expressions

A **regular expression** is a set of special characters that define a pattern.

A regular expression consists of two types of characters: literals and metacharacters.

A **literal** is just a character you wish to match in the target (i.e., the text that you are searching within).

A **metacharacter** is a special symbol that acts as a command to the regular expression parser (there are 14, listed below).

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

```
Example for phone numbers:   pattern="[+]?[0-9]{10,14}"
```

Regular Expression Syntax (ii)

In JavaScript, regular expressions are case sensitive and contained within forward slashes. For instance

let pattern = /ala/;

will find matches in these strings:

‘Salah Althobeiti’

‘Al malaz district'

Note: The provided annex document gives more details about REGEX with javascript.

Examples: “Salah Althobeiti".match(/ala/) , /ala/.test(“Salah Althobeiti")