COLLAGE OF COMPUTER SCIENCE & ENGINEERING

UNIVERSITY OF JEDDAH

جامعة جدة
University of Jeddah

كـلـيـة عـلــــــــوم و هـنـدســـة الحــــــاســـــب

جـــامعـــة جــــــــدة

JS: JavaScript For Web Development

CCSW 321 (Web Development)

What will be covered

- What is the Document Object Model (DOM)
- Accessing DOM elements
- Traversing DOM
- Manipulation DOM
- JS Events
- JS Data Validation
- Libraries and Frameworks
- Best Practices and Tips

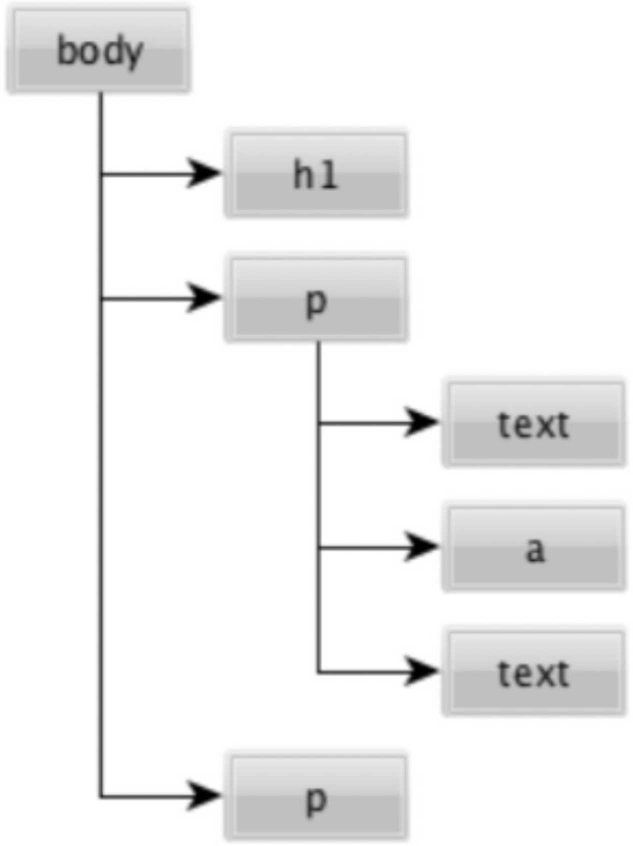
The Document Object Model (DOM)

- **What is the DOM?**
  - The Document Object Model is what allows web pages to **render**, **respond** to user events, and change.
  - The Document Object Model (DOM) is an **interface** that allows programs and scripts to **dynamically access** and **update** the content, structure, and style of a document.
  - When a web page is **loaded**, the **browser creates** a Document Object Model (DOM) of the page.

The Document Object Model (DOM)

- **What is the DOM?**

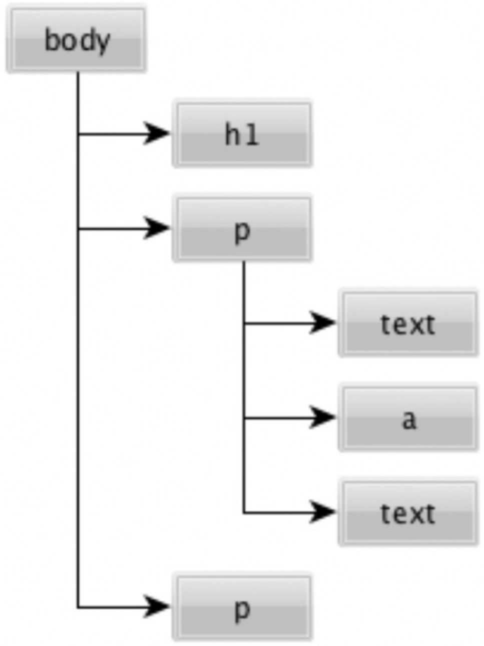
```
<body>

  <h1>Hello</h1>

  <p>

    Check out my

    <a href="/page">Page!</a>

    It's the best page out there
  </p>


  <p>Come back soon!</p>
</body>
```



HTML Code to **DOM** Representation

The Document Object Model (DOM)

- The **DOM represents** the **web page** as a **logical tree** where each branch of the tree ends in a node and each node contains objects.
- The **topmost** node is the **Document node**, followed by the **root** element node, and then the **child nodes** representing other elements and text nodes.
  - Each node can have 0 or 1 parent.
  - Each node can have 0 to many siblings/children nodes.



The Document Object Model (DOM)

- **DOM** Nodes:
  - Are **JavaScript objects**.
  - Have **Attributes** that are JavaScript **properties.**
  - They represent elements, attributes, and text in the HTML document.
- **DOM Attributes** define how the Node **looks** and **responds** to User **activity.**
- **DOM methods** allow programmatic access to the tree. With them, you can change the document's structure, style, and/or content.

The Document Object Model (DOM)

- The **document object** is the important connection between the DOM and JavaScript code.
- It Provides methods for navigating and manipulating the DOM.
- **What can we do with document object?**
    - Can **add**, **modify**, or **remove nodes** on the DOM, which will add, modify, or remove the corresponding element on the page.
    - Can **add**, **modify**, or **remove** the **attributes** of these nodes to change the attributes of an element.
    - Can **add**, **modify**, or **remove** existing **events** in the page.

The Document Object Model (DOM)

- We can **access elements** in the DOM by **searching the DOM** for HTML elements that match **certain criteria**.

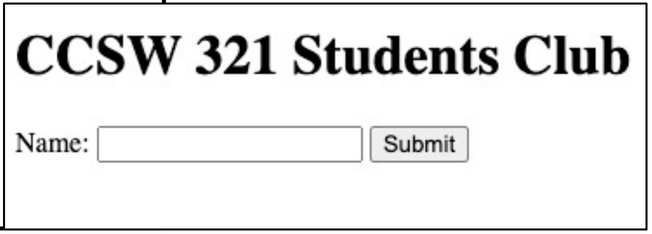
- The most used methods for accessing elements are:
  1 ◦ document.**getElementsByTagName**(‘nameOfTag’)
  2 ◦ document.**getElementsByName**('valueOfAttributeName ‘)
  3 ◦ document.**getElementsByClassName**(“valueOfAttributeClass”);
  4 ◦ document.**getElementById**(‘'valueOfAttributeId’)
  5 ◦ document.**querySelector**(‘cssSelector')
  6 ◦ document.**querySelectorAll**('cssSelector ')

- These methods help select the element we want to manipulate.

The Document Object Model (DOM)

- Let’s see how this works. Given the following HTML code:

```
<!DOCTYPE html>
<html>
  <head>…</head>
  <body>
     <h1 class='title'>CCSW 321 Students Club</h1>
    <form>
        <label>Student Name:</label>
        <input type='text' name='stdname' />
        <input type='submit’/>
        <p id='msg' ></p>
    </form>
  </body>
</html>
```



CCSW 321 Students Club

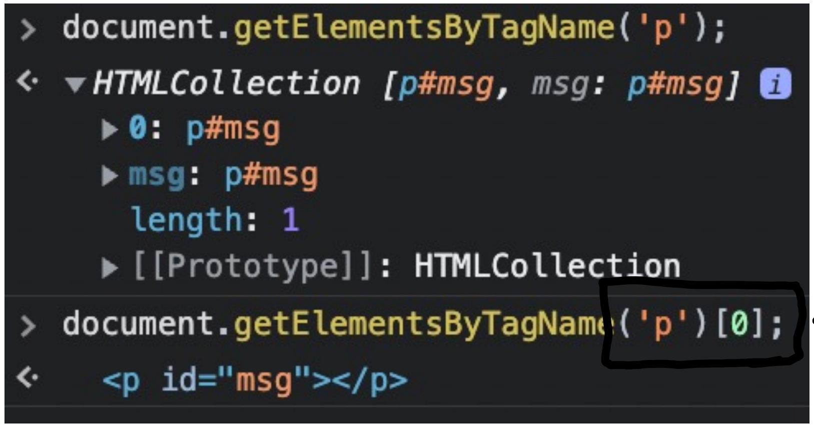
Name:

The Document Object Model (DOM)

```
let result = document.getElementsByTagName('p');
```

- **result** = will return a collection of all <p> tags found in the DOM tree. The **returned value** will always be a **collection** even if one <p> exists in the page.

- To access the first element, we use the index zero **result[0]**.

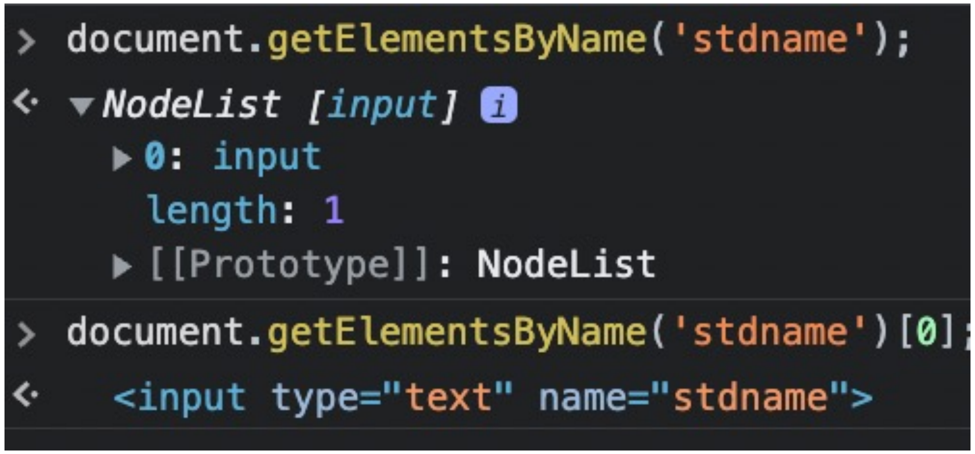


```
> document.getElementsByTagName('p');
<· ▾HTMLCollection [p#msg, msg: p#msg] i
     ▸0: p#msg
     ▸msg: p#msg
      length: 1
     ▸[[Prototype]]: HTMLCollection
> document.getElementsByTagName('p')[0];
<·   <p id="msg"></p>
```

The Document Object Model (DOM)

```
let result = document.getElementsByName('stdname');
```

- **result** = will return a nodeList of all HTML elements with **attribute 'name'** and **value 'stdname'.**
- To access the first element, we use the index zero **result[0]**.

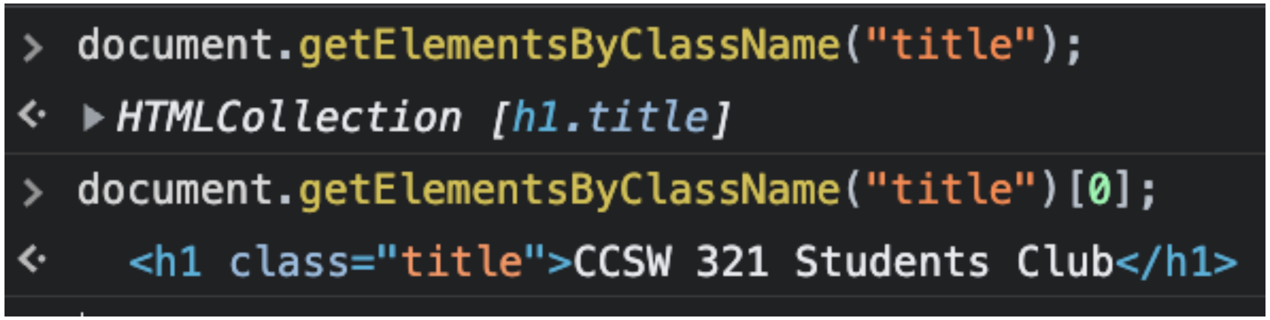


```
> document.getElementsByName('stdname');
<· ▼NodeList [input] i
    ▶0: input
     length: 1
    ▶[[Prototype]]: NodeList
> document.getElementsByName('stdname')[0];
<·   <input type="text" name="stdname">
```

The Document Object Model (DOM)

```
let result = document.getElementsByClassName('msg');
```

- **result** = will return a collection of all elements with **attribute ‘class’** and **value ‘title’.**
- To access the first element, we use the index zero result[0].



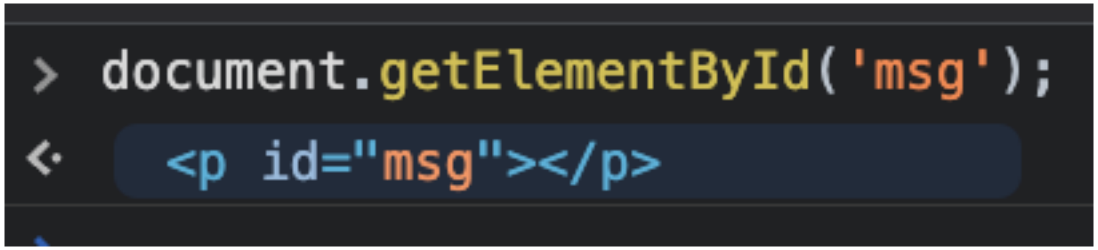
```
> document.getElementsByClassName("title");
<· ▶ HTMLCollection [h1.title]
> document.getElementsByClassName("title")[0];
<·   <h1 class="title">CCSW 321 Students Club</h1>
```

The Document Object Model (DOM)

```
let result = document.getElementById('msg');
```

- **result** = will return a single node, the **first** element with the **attribute ‘id’** and **value ‘msg’.**

- Unlike previous methods, this one returns a single node.



```
> document.getElementById('msg');
<·   <p id="msg"></p>
```

The Document Object Model (DOM)

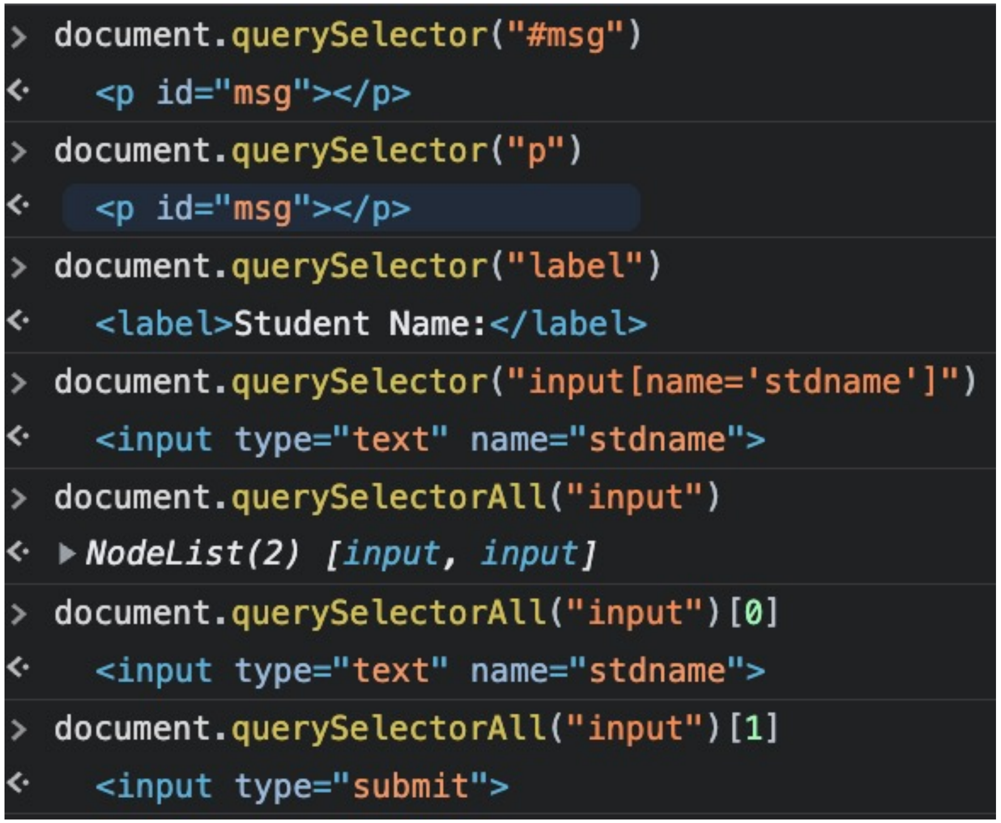
```
let result = document.querySelector(‘#msg');
```

- **result** = will return a single node, the **first** element that with attribute **id** and **value “msg”.**
- **querySelector** is a powerful method. You can use your knowledge of CSS Selectors to find any element in the DOM.
  - **querySelector(".title")** = will select the first element with class name ‘title’.
  - **querySelector("p")** = will select the first <p> element.
  - **querySelector("input[name='stdname’]”)** = will select the first <input> with attribute ‘name’ and value ‘stdname’.

The Document Object Model (DOM)

```
let result = document.querySelectorAll(‘input');
```

- **result** = will return a NodeList of all existing <input> elements.
- **querySelectorAll** works in similar manner as querySelector but returns all elements matching criteria instead of the first element only.



```
> document.querySelector("#msg")
<·   <p id="msg"></p>
> document.querySelector("p")
<·   <p id="msg"></p>
> document.querySelector("label")
<·   <label>Student Name:</label>
> document.querySelector("input[name='stdname']")
<·   <input type="text" name="stdname">
> document.querySelectorAll("input")
<· ▸NodeList(2) [input, input]
> document.querySelectorAll("input")[0]
<·   <input type="text" name="stdname">
> document.querySelectorAll("input")[1]
<·   <input type="submit">
```

The Document Object Model (DOM)

HTMLColelction and NodeList

- Most of the DOM selector methods will return a **collection of Nodes**, which is an object called **HTMLCollection** or **NodeList**.
- The **NodeList** and **HTMLCollection** are objects, **not arrays**, which means they do not have access to the usual array properties or methods.
- In most cases, you do not need to convert HTMLCollection or NodeList into an array, but in case you did, then to convert them into arrays, use the following:

```
Let result = Array.from(document.getElementsByTagName('input'))
```

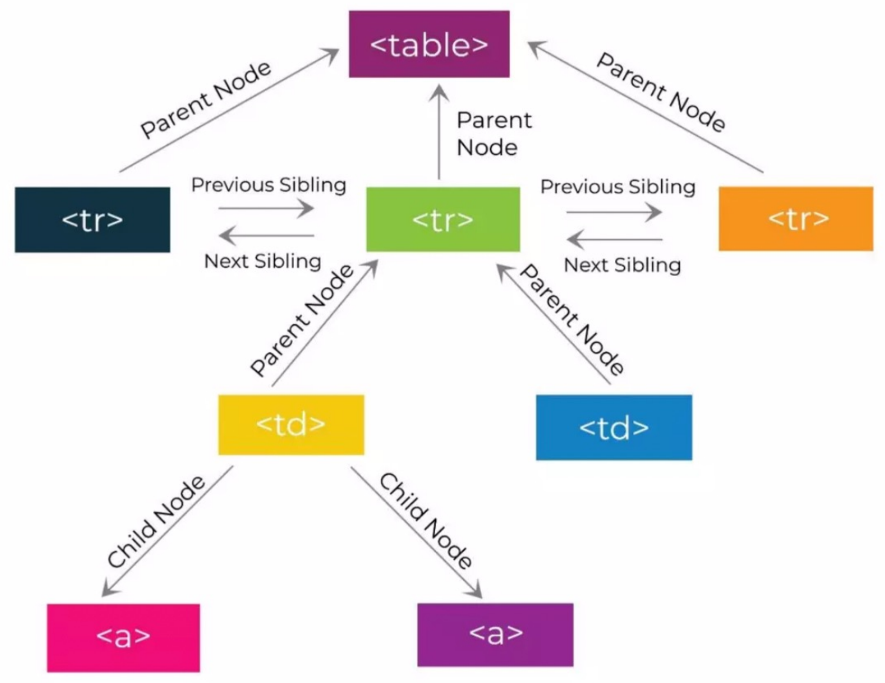
- Now **result** is an Array with access to array properties and methods.

The Document Object Model (DOM)

- **Traversing the DOM** tree means navigating through the hierarchy of nodes in the Document Object Model (DOM) tree.
- In the DOM tree, each element is represented as a node, and each node has a **specific relationship** to other nodes in the tree. The relationship between nodes can be categorized as **parent**, **child**, **sibling**, **ancestor**, or **descendant**.
- By using DOM traversal, developers can access and manipulate elements that may be hidden or difficult to access using other methods.

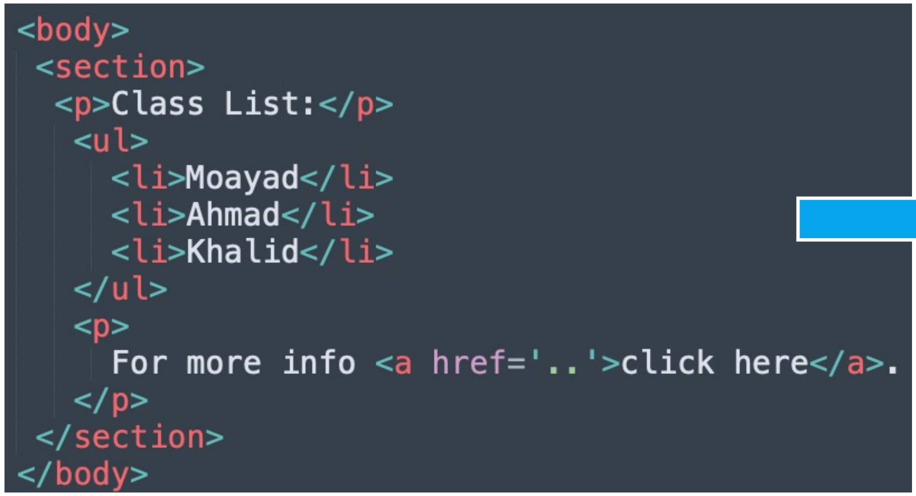
The Document Object Model (DOM)

- **Traversing the DOM** tree involves selecting an initial node and then moving to other nodes based on their relationship to the initial node. This can be done using DOM traversal methods:
  1 ◦ **parentNode**
  2 ◦ **childNodes[i]**
  3 ◦ **firstChild**
  4 ◦ **lastChild**
  5 ◦ **previousSibling**
  6 ◦ **nextSibling**

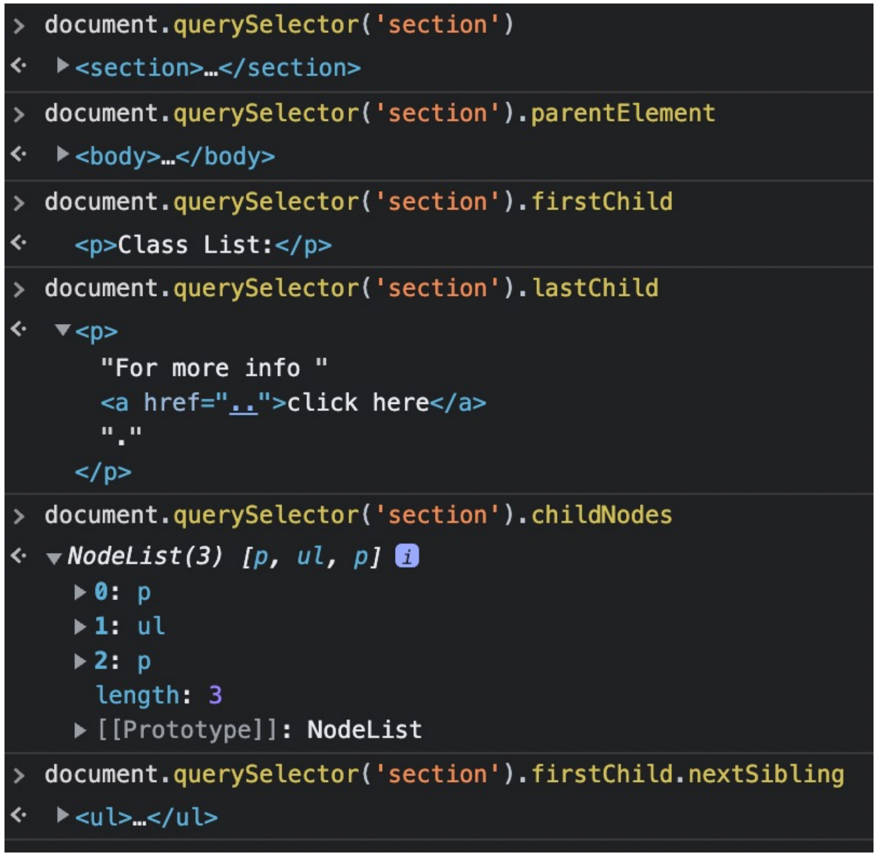


The Document Object Model (DOM)

- Let's see how this works.



```
<body>
  <section>
   <p>Class List:</p>
    <ul>
      <li>Moayad</li>
      <li>Ahmad</li>
      <li>Khalid</li>
    </ul>
    <p>
      For more info <a href='..'>click here</a>.
    </p>
  </section>
</body>
```



```
> document.querySelector('section')
<·  ▶<section>…</section>
> document.querySelector('section').parentElement
<·  ▶<body>…</body>
> document.querySelector('section').firstChild
<·   <p>Class List:</p>
> document.querySelector('section').lastChild
<·  ▼<p>
       "For more info "
       <a href="..">click here</a>
       "."
     </p>
> document.querySelector('section').childNodes
<· ▼NodeList(3) [p, ul, p] i
    ▶0: p
    ▶1: ul
    ▶2: p
     length: 3
    ▶[[Prototype]]: NodeList
> document.querySelector('section').firstChild.nextSibling
<·  ▶<ul>…</ul>
```

The Document Object Model (DOM)

- Modifying elements in the DOM allows us to:
  - Add/modify/remove HTML **elements**.
  - Add/modify/remove HTML **attributes**, for example:
    - Update an image src attribute.
    - Update the ’type’ of an input which, e.g, allows us to enable/disable elements.
  - Add/modify/remove **CSS styling**, for example:
    - Hide/show elements by updating the “display” property.
  - Attach/detach **event listeners**.

The Document Object Model (DOM)

- Methods to create/remove nodes:
  - **createElement**(tagName) = will create a new node that corresponds to the specified **tagName**.
  - parentNode.**removeChild**(oldNode) = will remove the **oldNode** from the **parentNode.**
- When removing a node, sometimes it is challenges to identify the parentNode, so there's a common trick used to remove nodes:
  - oldNode.parentNode.removeChild(oldNode)

We select the node to be removed, and then we invoke "parentNode" property to select its parent, and then we remove the node.

The Document Object Model (DOM)

- Methods to add nodes to the page (Linking to page):
  - **appendChild**(newNode): Insert newNode at end of current Node
  - **prependChild**(newNode): Insert newNode at beginning of current Node
  - **insertBefore**(newNode, siblingNode): Insert newNode before a certain childNode.
- Once nodes are created, they are just free floating, and **not connected** to the document itself until you **link them** to the DOM. You must link them to view them on page.

The Document Object Model (DOM)

Properties to edit content:

- element.**innerHTML** = gets/sets the HTML content of a node. Used with elements without a value attribute, e.g., <div>, <p>.
- element.**innerText** = similar to innerHTML but gets/sets the textual content only.
- element.**value** = gets/sets the value element of a form element. Used with form elements mostly, e.g., <input>, <select>, etc.
- element.**setAttribute**(name,value) = sets the attribute to given value.

The Document Object Model (DOM)

Creating elements in the DOM:

- You can create nodes easily using the tagName as follows:

```
// create a <p> tag
let ptag = document.createElement(‘p’);
```

- Nodes are just free floating, and not connected to the document itself until you link them to the DOM.

```
// add ptag to the page
document.querySelector(‘body’).appendChild(ptag);
```

The Document Object Model (DOM)

Removing elements from the DOM:

- To remove a node element, we can do the following:

```
<body> <p>Hello World!</p> </body>
```

- Removing the <p> tag along with its content

```
const pNode = document.querySelector(‘p’);
   pNode.parentNode.removeChild(pNode);
```

The Document Object Model (DOM)

Modifying elements in the DOM:

- We can modify the textual content of a <p>:

```
const pNode = document.querySelector('p');
        pNode.innerHTML = ‘new Text’;
```

```
const pNode = document.querySelector('p');
        pNode.innerText = ‘new Text’;
```

- The main difference is that **innerHTML** returns the HTML content inside an element, including any tags, while **innerText** returns only the text content inside an element, without any tags..

The Document Object Model (DOM)

Modifying elements in the DOM:

- You can access attributes of an HTML element via a property (field) of the DOM object, e.g., change an image:

```
    const image = document.querySelector('img');
            image.src = 'new-picture.png';
```

- You can do the same using setAttribute:

```
    const image = document.querySelector('img');
      image.setAttribute('src','new-picture.png');
```

The Document Object Model (DOM)

Modifying CSS of elements in the DOM:

- We can use the **style** property to modify CSS:

```
const element = document.querySelector(‘p');

      element.style.fontWeight = “bold”;
```

Note the following:

background-color $\longrightarrow$	backgroundColor
font-weight $\longrightarrow$	fontWeight

The Document Object Model (DOM)

Modifying CSS of elements in the DOM:

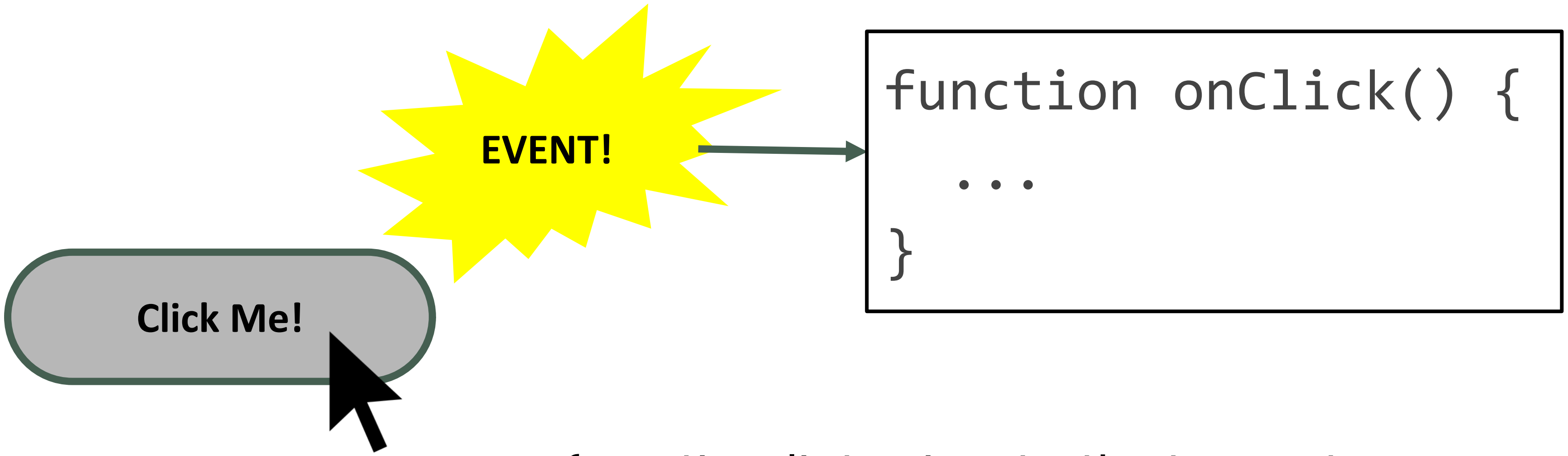
- Alternatively, we can update the class name:

```
//add a CSS class
document.getElementById(“MyElement”).classList.add(‘class’);

//remove a CSS class
document.getElementById(“MyElement”).classList.remove(‘class’);
```

JavaScript Events

- JavaScript in the browser is mostly even-driven.

  The code doesn't run right away, but it executes after some event fires.



```
function onClick() {
   ...
}
```

Any function listening to that event now executes. This function is called an "**event handler.**"

JavaScript Events

- Events can be **attached** to DOM **elements** so that when a certain **event occurs**, they can **invoke** a given **function**.
- Examples of commonly used events are:
  - **Submit** = fires when form is submitted.
  - **Click** = fires when element is clicked.
  - **Keypress** = fires when a keyboard button is pressed.
  - **Focus** = fires An element gets focus.
- For a full list of possible events, please refer to https://www.w3schools.com/jsref/dom_obj_event.asp

JavaScript Events

Event handling with the DOM:

- Each DOM object has the following functiond:

```
addEventListener(event name, function name);
```

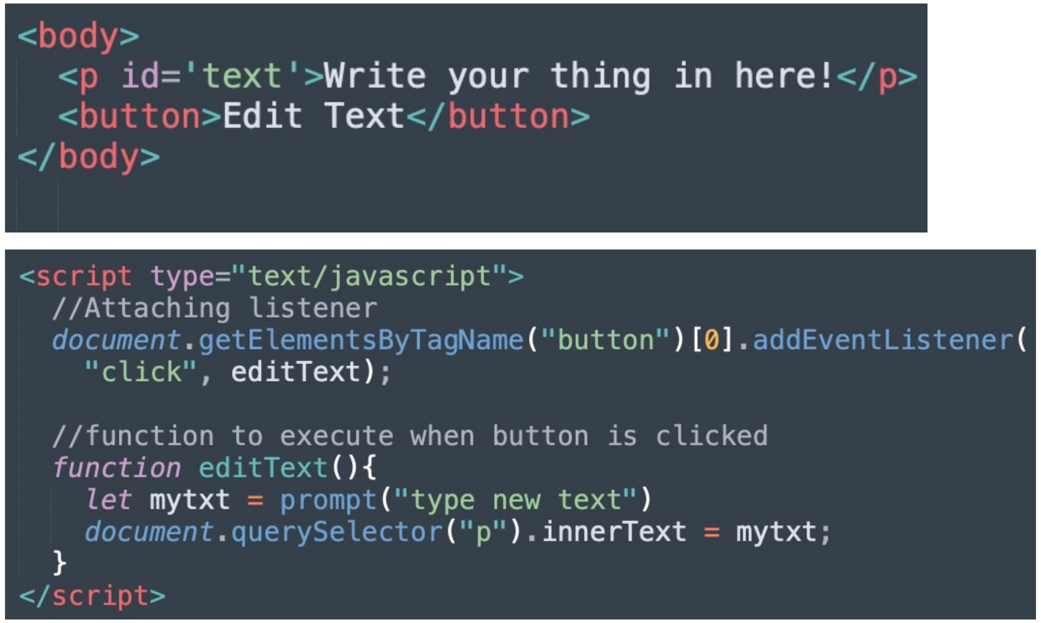
```
removeEventListener(event name, function name);
```

- **event name** is the string name of the JavaScript event you want to listen to (e.g. click, focus, blur, etc.)

- **function name** is the name of the JavaScript function you want to execute when the event fires

JavaScript Events

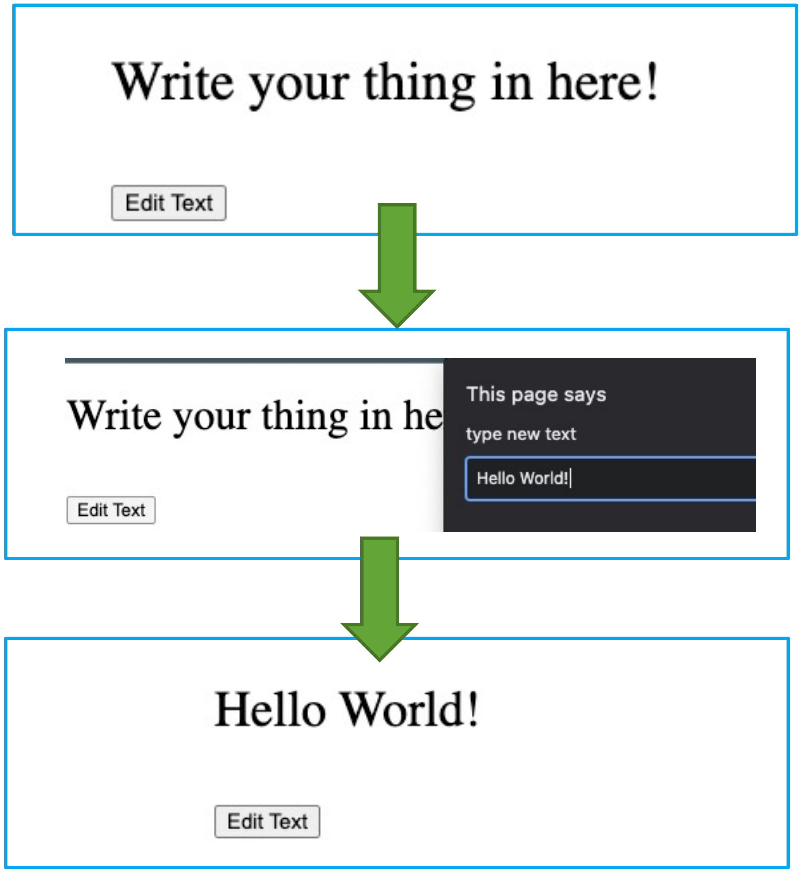
CODE



```
<body>
  <p id='text'>Write your thing in here!</p>
  <button>Edit Text</button>
</body>
```

```
<script type="text/javascript">
  //Attaching listener
  document.getElementsByTagName("button")[0].addEventListener(
    "click", editText);

  //function to execute when button is clicked
  function editText(){
    let mytxt = prompt("type new text")
    document.querySelector("p").innerText = mytxt;
  }
</script>
```

OUTPUT



Write your thing in here!

Edit Text

Write your thing in he

Edit Text

This page says

type new text

Hello World!

Hello World!

Edit Text

JavaScript DOM/Events Example

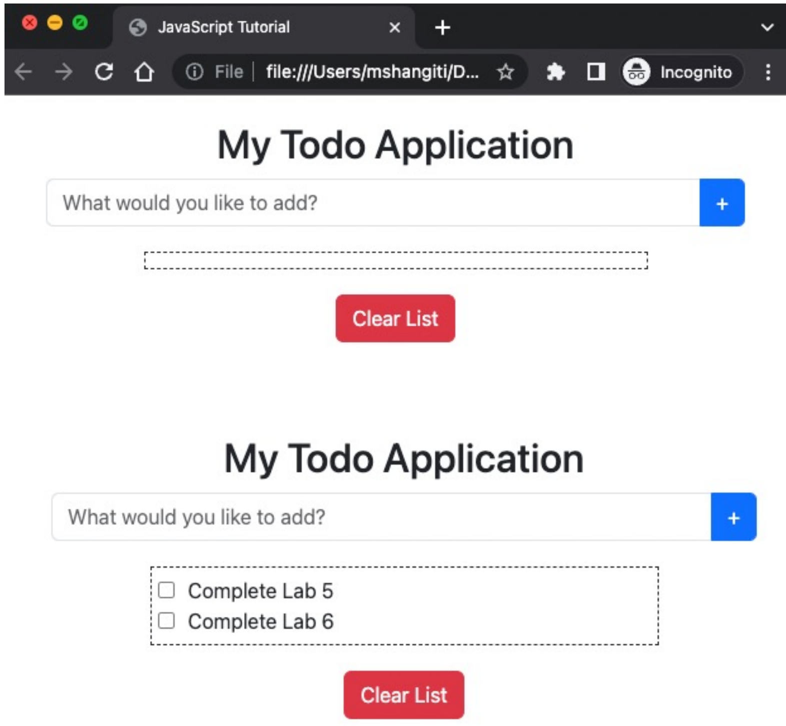
- Can you create this simple app?

Step0: Build the HTML/CSS needed. In this example we used Bootstrap to improve the design, but feel free to use your own styling.

Step1: Attach an event listener to the add button to call 'addElement()' function on mouse click.

Step2: Attach an event listener to call 'addElement()' function on keyboard 'enter' press.

Step3: Attach an event listener to the 'clear List' button to call 'emptyList()' function on mouse click.

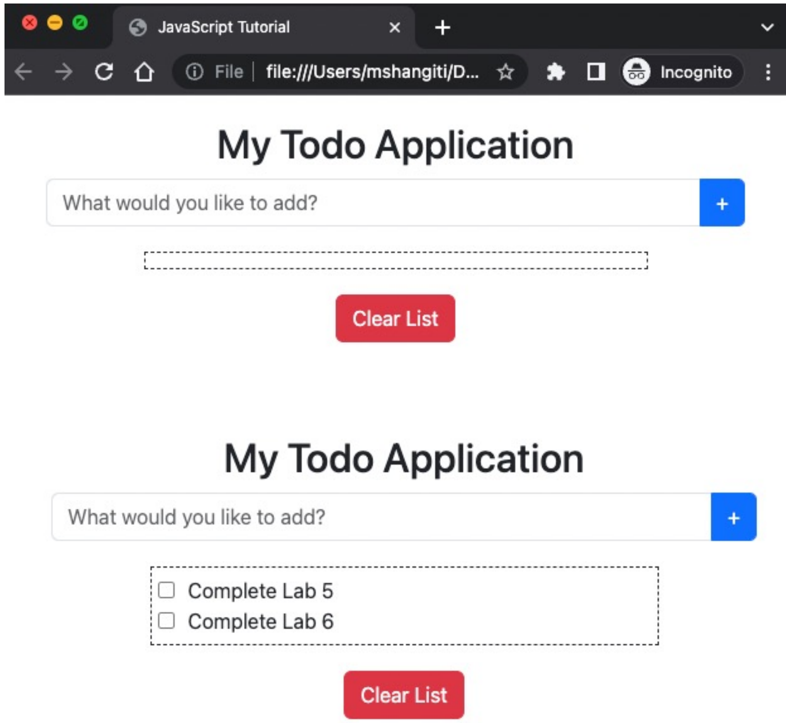


JavaScript DOM/Events Example

- Can you create this simple app?

Step4. Create the addElement() function that would read user input and add it to the list. Attach an event listener to added element to remove the item when the 'checkbox' is clicked.

Step5. Create the emptyList() function that would remove all items from the list.



JavaScript DOM/Events Example

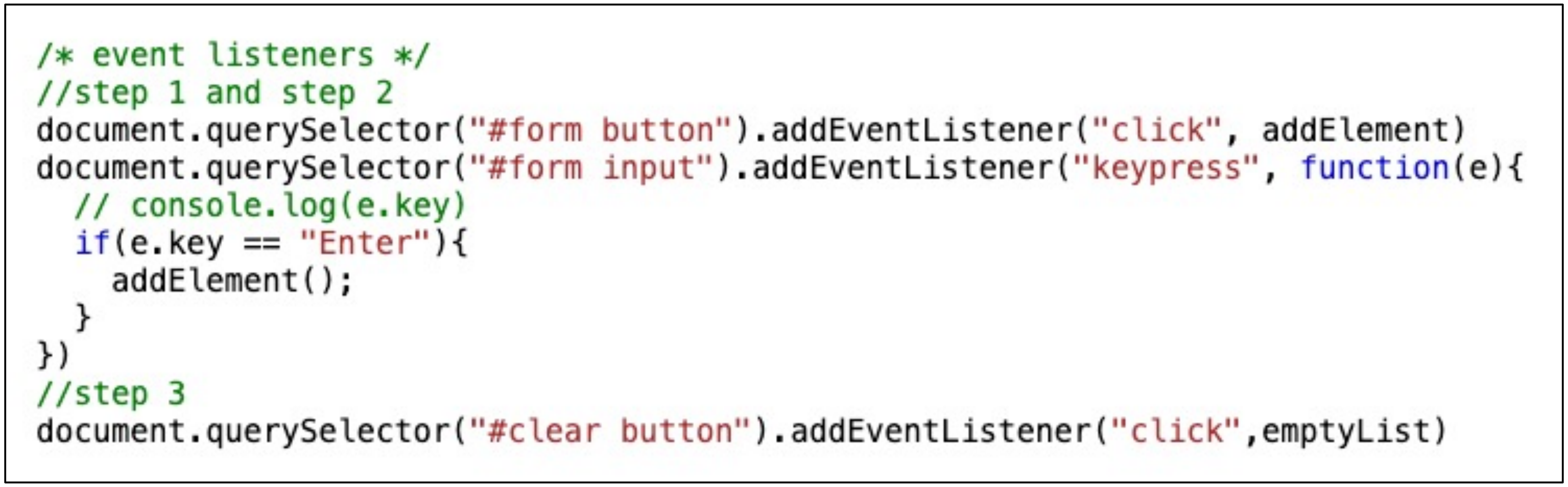
- Step 0: HTML and CSS

```
<!DOCTYPE html>
<html>
  <head>...</head>
  <body>
   <section>
      <section id='form'>
        <h1>My Todo Application</h1>
        <div class='input-group'>
        <input class='form-control' type="text" name="item" placeholder="What would you like to add?">
        <button  class='btn btn-primary'>+</button>
        </div>
      </section>
      <section id='mylist'>
      </section>
      <section id='clear'>
        <button  class='btn btn-danger'>Clear List</button>
      </section>
    </section>
  </body>
</html>
```

```
<style>
  body{
    text-align: center;
  }
  section{
    margin: 20px;
  }

  label{
    margin-left: 10px;
  }

  #mylist{
    text-align: left;
    width: 400px;
    margin:0 auto;
    border:1px dashed;
    padding: 1%;
    cursor:pointer;
  }
</style>
```

JavaScript DOM/Events Example

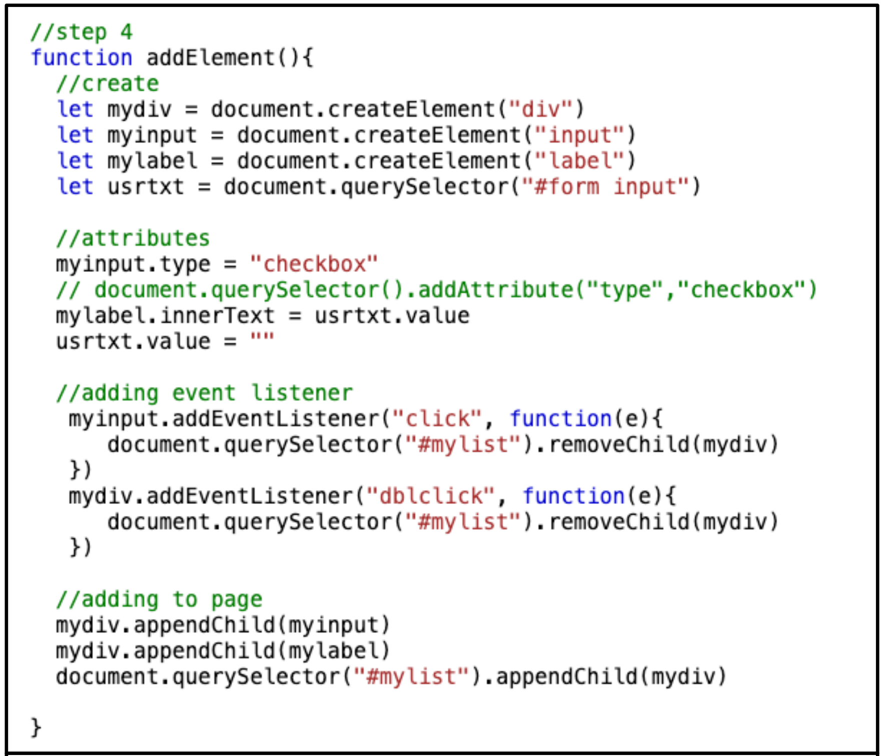
- Step 1, Step 2, and Step 3: Event Listeners



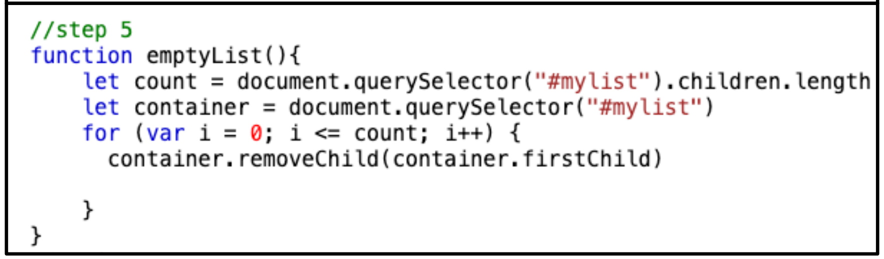
```
/* event listeners */
//step 1 and step 2
document.querySelector("#form button").addEventListener("click", addElement)
document.querySelector("#form input").addEventListener("keypress", function(e){
  // console.log(e.key)
  if(e.key == "Enter"){
    addElement();
  }
})
//step 3
document.querySelector("#clear button").addEventListener("click",emptyList)
```

JavaScript DOM/Events Example

- Step 4 and Step 5: Functions



```
//step 4
function addElement(){
  //create
  let mydiv = document.createElement("div")
  let myinput = document.createElement("input")
  let mylabel = document.createElement("label")
  let usrtxt = document.querySelector("#form input")

  //attributes
  myinput.type = "checkbox"
  // document.querySelector().addAttribute("type","checkbox")
  mylabel.innerText = usrtxt.value
  usrtxt.value = ""

  //adding event listener
   myinput.addEventListener("click", function(e){
      document.querySelector("#mylist").removeChild(mydiv)
   })
   mydiv.addEventListener("dblclick", function(e){
      document.querySelector("#mylist").removeChild(mydiv)
   })

  //adding to page
  mydiv.appendChild(myinput)
  mydiv.appendChild(mylabel)
  document.querySelector("#mylist").appendChild(mydiv)

}
```



```
//step 5
function emptyList(){
    let count = document.querySelector("#mylist").children.length
    let container = document.querySelector("#mylist")
    for (var i = 0; i <= count; i++) {
      container.removeChild(container.firstChild)

    }
}
```

JavaScript Data Validation Layer

- **User data validation** is the process of checking whether the data entered by the user into a web form is **valid**, **correct**, and **consistent** with the **expected format**, **data type**, **range**, and other requirements.

- **Why** is it **important**?
  - It helps to **prevent errors**, **inconsistencies**, and **security breaches** that could result from incorrect or malicious data inputs.
  - It **enhances the user experience** by providing immediate feedback on incorrect input and **reducing the risk of errors and user frustration**.

JavaScript Data Validation Layer

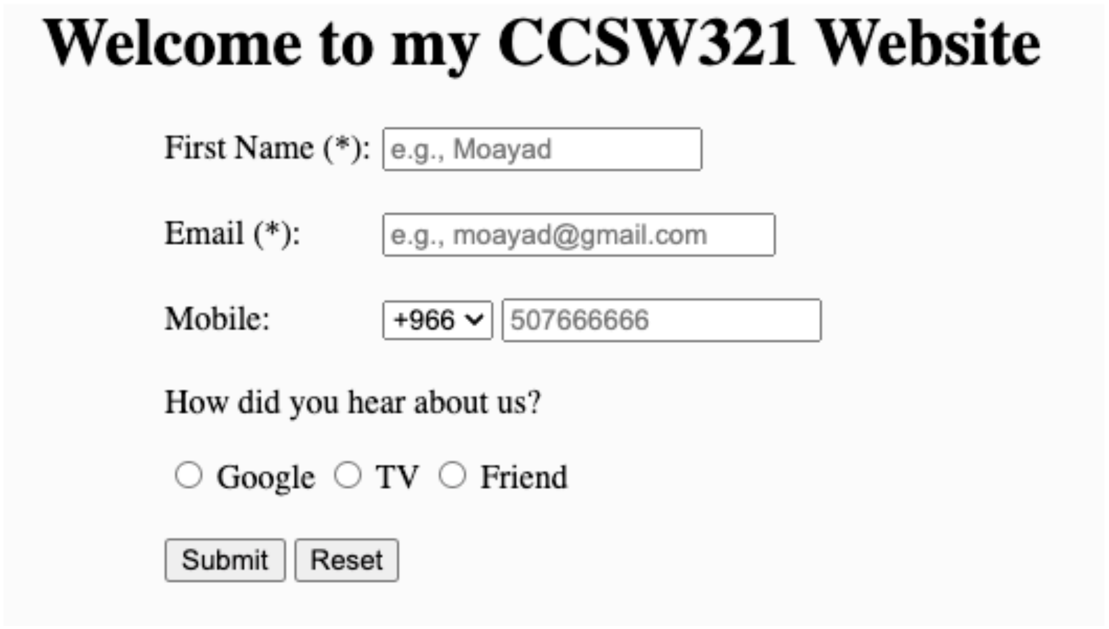
- What to Check for in User Data Validation?
  - **Presence validation**: Ensuring that the input is not empty or null
  - **Data type validation**: correct data type (e.g., string, number, boolean)?
  - **Format validation**: follows the specified format (e.g., email, phone number, date)?
  - **Range validation**: falls within a specified range (e.g., age, price, quantity)?
  - **Length validation**: Does not exceed a specific length limit (e.g., password)?
  - **Whitelist Validation**: is one of the accepted values (e.g., gender)?

JavaScript Data Validation Layer

- JavaScript provides various methods and techniques for validating user input data.
  - Using **regular expressions** to check for the format of the input.
  - Using **built-in functions** such as isNaN() to check for numeric input.
  - Using **if-else statements** and loops to check for specific conditions and requirements.
  - Using **third-party libraries** and frameworks such as jQuery or React that offer built-in validation methods.

JavaScript Data Validation Layer

- Let's see an example of how this works.
- Can you build this form using HTML?



Welcome to my CCSW321 Website

First Name (*): e.g., Moayad

Email (*): e.g., moayad@gmail.com

Mobile: +966 507666666

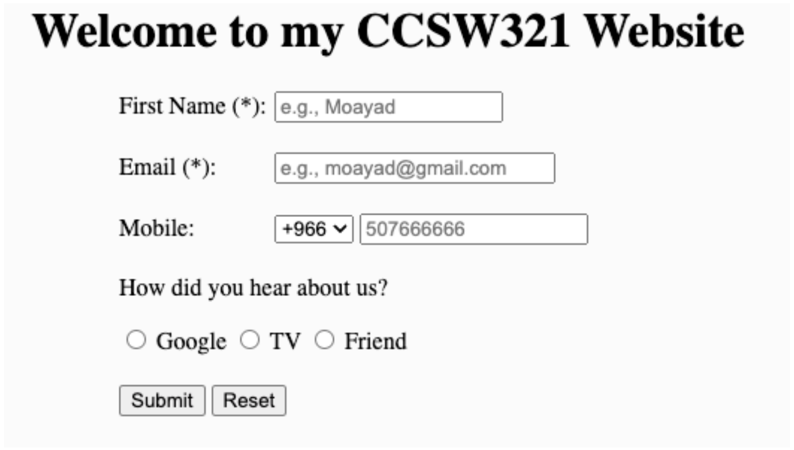
How did you hear about us?

○ Google ○ TV ○ Friend

What kind of checks do you think we need to do?

JavaScript Data Validation Layer

- Examples of validation checks:
  - Does name contain English letters only?
  - Is the name within expected length, e.g., between 2 and 100 chars?
  - Is the email in the expected format?
  - Mobile contains only numbers?
  - Mobile is within expected range?
  - Was the selection of "how the user heard about us" from the list provided (code can be injected!)?
  - Etc.



Welcome to my CCSW321 Website

First Name (*): e.g., Moayad

Email (*): e.g., moayad@gmail.com

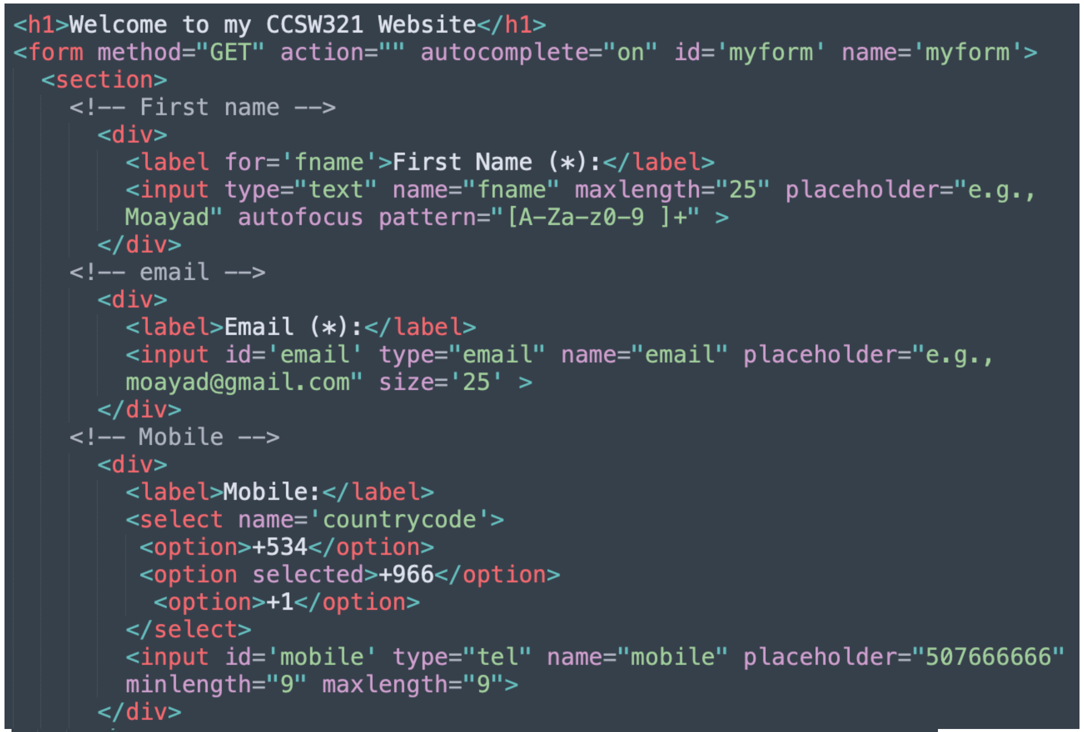
Mobile: +966 507666666

How did you hear about us?

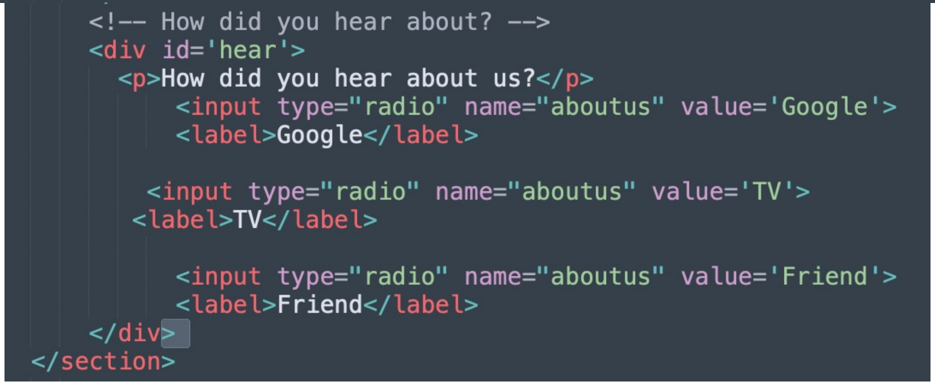
○ Google ○ TV ○ Friend

JavaScript Data Validation Layer

- HTML:



```html
<h1>Welcome to my CCSW321 Website</h1>
<form method="GET" action="" autocomplete="on" id='myform' name='myform'>
  <section>
    <!-- First name -->
      <div>
        <label for='fname'>First Name (*):</label>
        <input type="text" name="fname" maxlength="25" placeholder="e.g.,
        Moayad" autofocus pattern="[A-Za-z0-9 ]+" >
      </div>
    <!-- email -->
      <div>
        <label>Email (*):</label>
        <input id='email' type="email" name="email" placeholder="e.g.,
        moayad@gmail.com" size='25' >
      </div>
    <!-- Mobile -->
      <div>
        <label>Mobile:</label>
        <select name='countrycode'>
         <option>+534</option>
         <option selected>+966</option>
          <option>+1</option>
        </select>
        <input id='mobile' type="tel" name="mobile" placeholder="507666666"
        minlength="9" maxlength="9">
      </div>
```

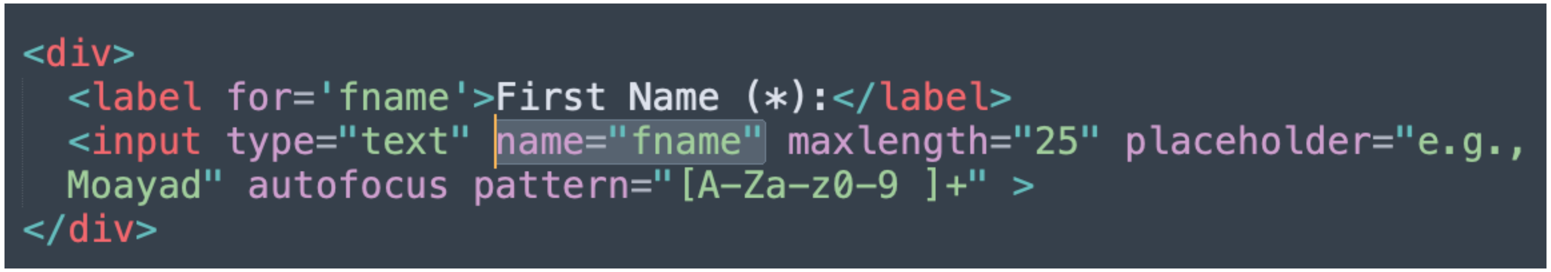


```html
      <!-- How did you hear about? -->
      <div id='hear'>
        <p>How did you hear about us?</p>
            <input type="radio" name="aboutus" value='Google'>
            <label>Google</label>

          <input type="radio" name="aboutus" value='TV'>
         <label>TV</label>

            <input type="radio" name="aboutus" value='Friend'>
            <label>Friend</label>
      </div>
  </section>
```

JavaScript Data Validation Layer

- We need to make sure each input field has a ‘name’ attribute. Without this attribute, nothing will be sent to the backend.

- The name attribute will act as a unique identifier that allows us to access the user data using DOM.

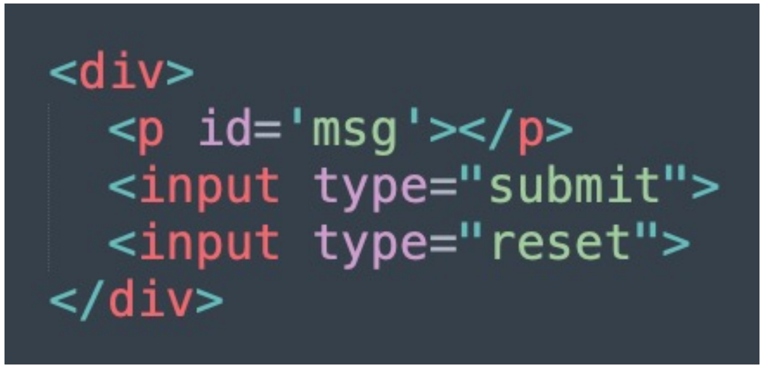


```
<div>
  <label for='fname'>First Name (*):</label>
  <input type="text" name="fname" maxlength="25" placeholder="e.g.,
  Moayad" autofocus pattern="[A-Za-z0-9 ]+" >
</div>
```

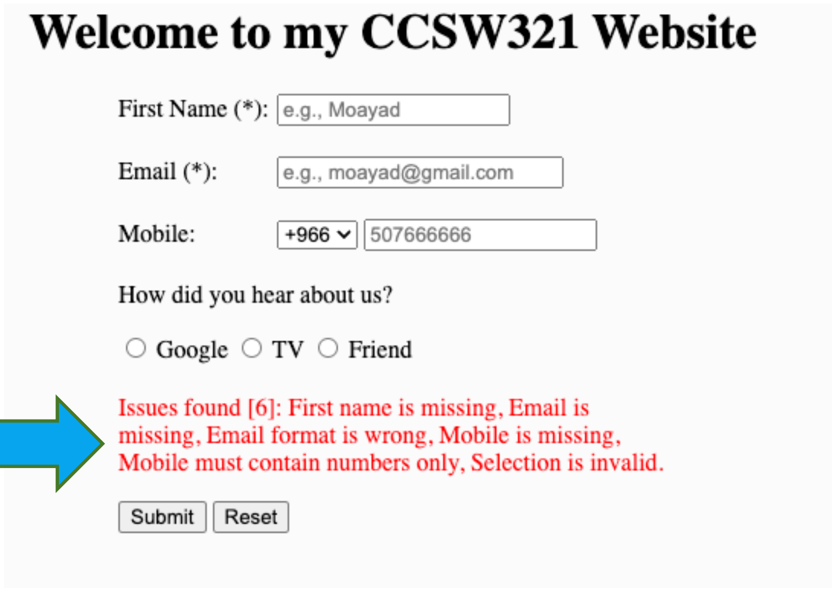
```
let fname = document.getElementsByName('fname')[0].value;
```

JavaScript Data Validation Layer

- In order to show the **error message** back to the user, we need to either user an alert() box, or insert/update an HTML tag to the <body> that shows the error messages.



```
<div>
  <p id='msg'></p>
  <input type="submit">
  <input type="reset">
</div>
```



Welcome to my CCSW321 Website

First Name (*):	e.g., Moayad
Email (*):	e.g., moayad@gmail.com
Mobile:	+966 507666666

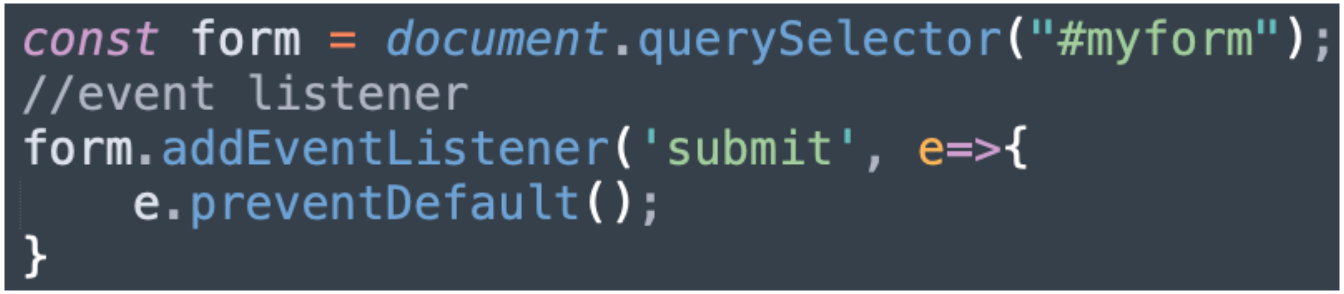
How did you hear about us?

○ Google ○ TV ○ Friend

Issues found [6]: First name is missing, Email is missing, Email format is wrong, Mobile is missing, Mobile must contain numbers only, Selection is invalid.

JavaScript Data Validation Layer

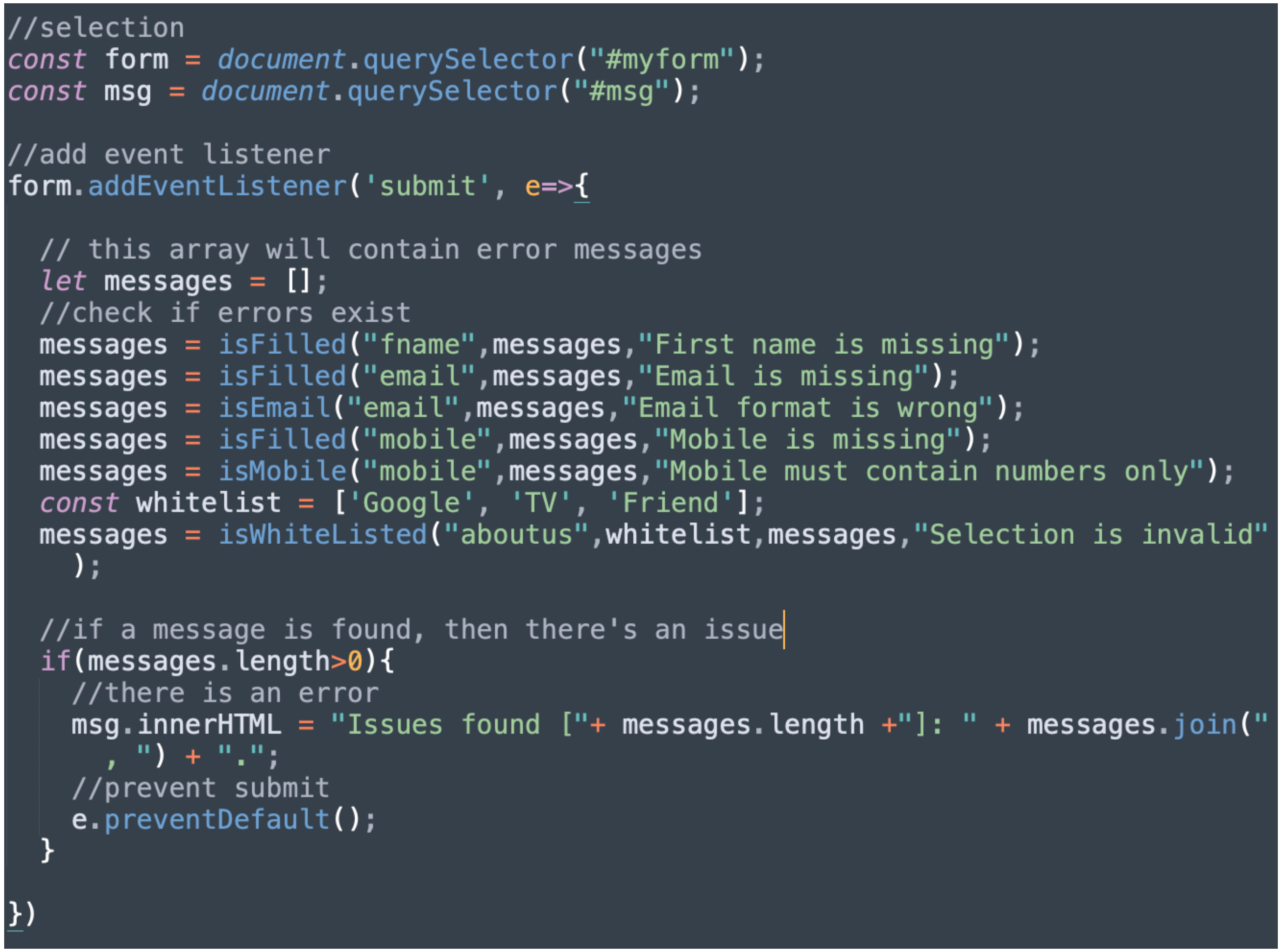
- The JS frontend validation occurs after the user has clicked on ‘submit’ and before the data is sent to the backend.
- In order to process the user data at the frontend, we need to stop the form from sending the data to the backend (**prevent submission**).
- To accomplish this, we need to **attach an event listener** and **stop the form from submission**.



```
const form = document.querySelector("#myform");
//event listener
form.addEventListener('submit', e=>{
    e.preventDefault();
}
```

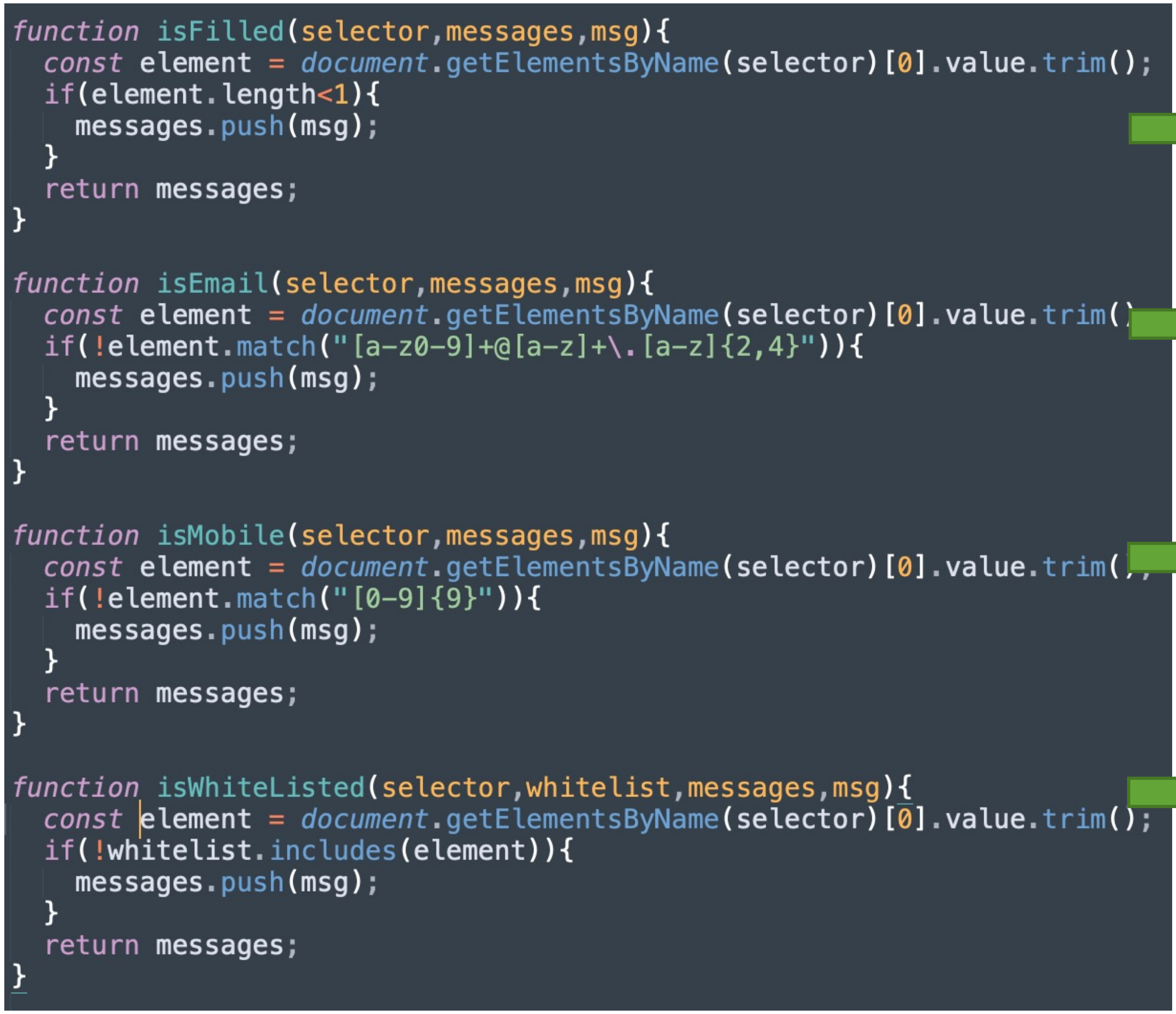
JavaScript Data Validation Layer

- We then add our JS code:



```
//selection
const form = document.querySelector("#myform");
const msg = document.querySelector("#msg");

//add event listener
form.addEventListener('submit', e=>{

  // this array will contain error messages
  let messages = [];
  //check if errors exist
  messages = isFilled("fname",messages,"First name is missing");
  messages = isFilled("email",messages,"Email is missing");
  messages = isEmail("email",messages,"Email format is wrong");
  messages = isFilled("mobile",messages,"Mobile is missing");
  messages = isMobile("mobile",messages,"Mobile must contain numbers only");
  const whitelist = ['Google', 'TV', 'Friend'];
  messages = isWhiteListed("aboutus",whitelist,messages,"Selection is invalid"
    );

  //if a message is found, then there's an issue
  if(messages.length>0){
    //there is an error
    msg.innerHTML = "Issues found ["+ messages.length +"]: " + messages.join("
      , ") + ".";
    //prevent submit
    e.preventDefault();
  }

})
```

JavaScript Data Validation Layer



```
function isFilled(selector,messages,msg){
  const element = document.getElementsByName(selector)[0].value.trim();
  if(element.length<1){
    messages.push(msg);
  }
  return messages;
}

function isEmail(selector,messages,msg){
  const element = document.getElementsByName(selector)[0].value.trim()
  if(!element.match("[a-z0-9]+@[a-z]+\.[a-z]{2,4}")){
    messages.push(msg);
  }
  return messages;
}

function isMobile(selector,messages,msg){
  const element = document.getElementsByName(selector)[0].value.trim();
  if(!element.match("[0-9]{9}")){
    messages.push(msg);
  }
  return messages;
}

function isWhiteListed(selector,whitelist,messages,msg){
  const element = document.getElementsByName(selector)[0].value.trim();
  if(!whitelist.includes(element)){
    messages.push(msg);
  }
  return messages;
}
```

Presence validation

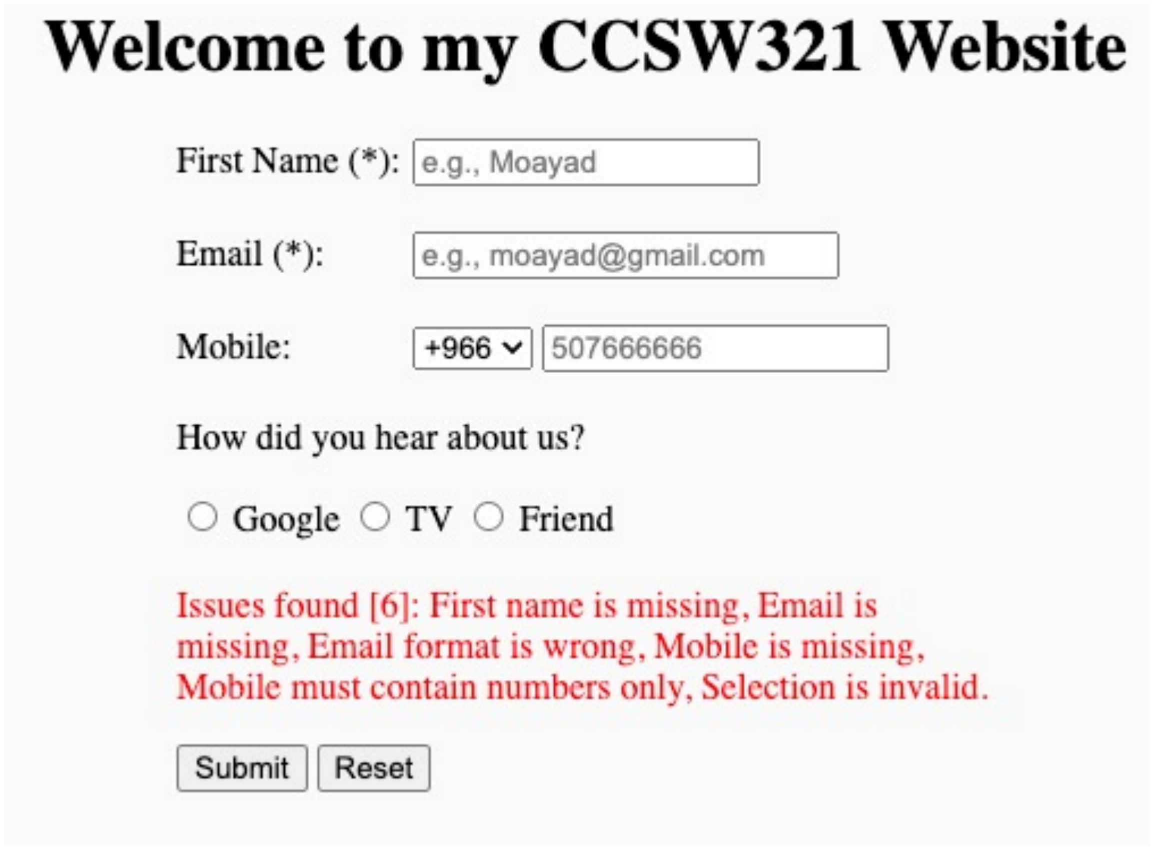
Format Validation

Data type and Length

Whitelist Validation

JavaScript Data Validation Layer

- Input = No entries
- Output:



Welcome to my CCSW321 Website

First Name (*): e.g., Moayad

Email (*): e.g., moayad@gmail.com

Mobile: +966 507666666

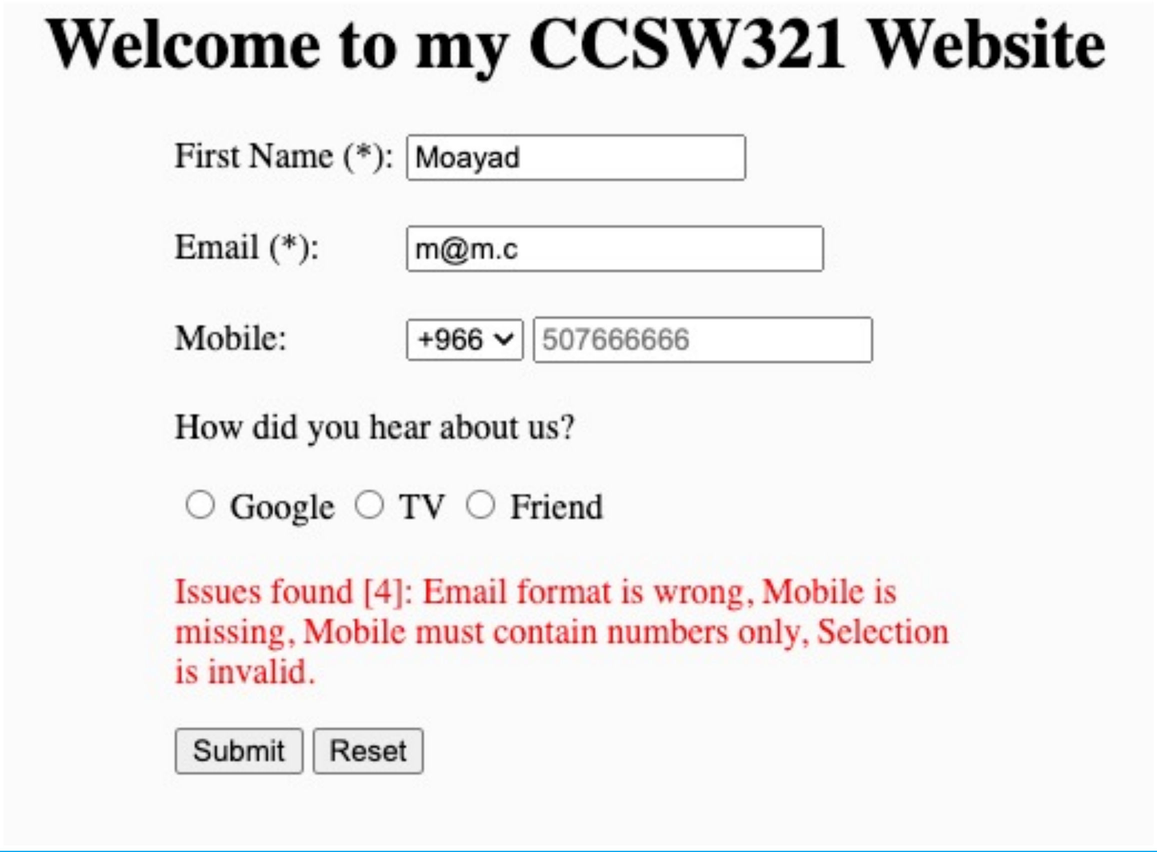
How did you hear about us?

○ Google ○ TV ○ Friend

Issues found [6]: First name is missing, Email is missing, Email format is wrong, Mobile is missing, Mobile must contain numbers only, Selection is invalid.

JavaScript Data Validation Layer

- Input = **invalid** email
- Output:



Welcome to my CCSW321 Website

First Name (*):	Moayad
Email (*):	m@m.c
Mobile:	+966 507666666

How did you hear about us?

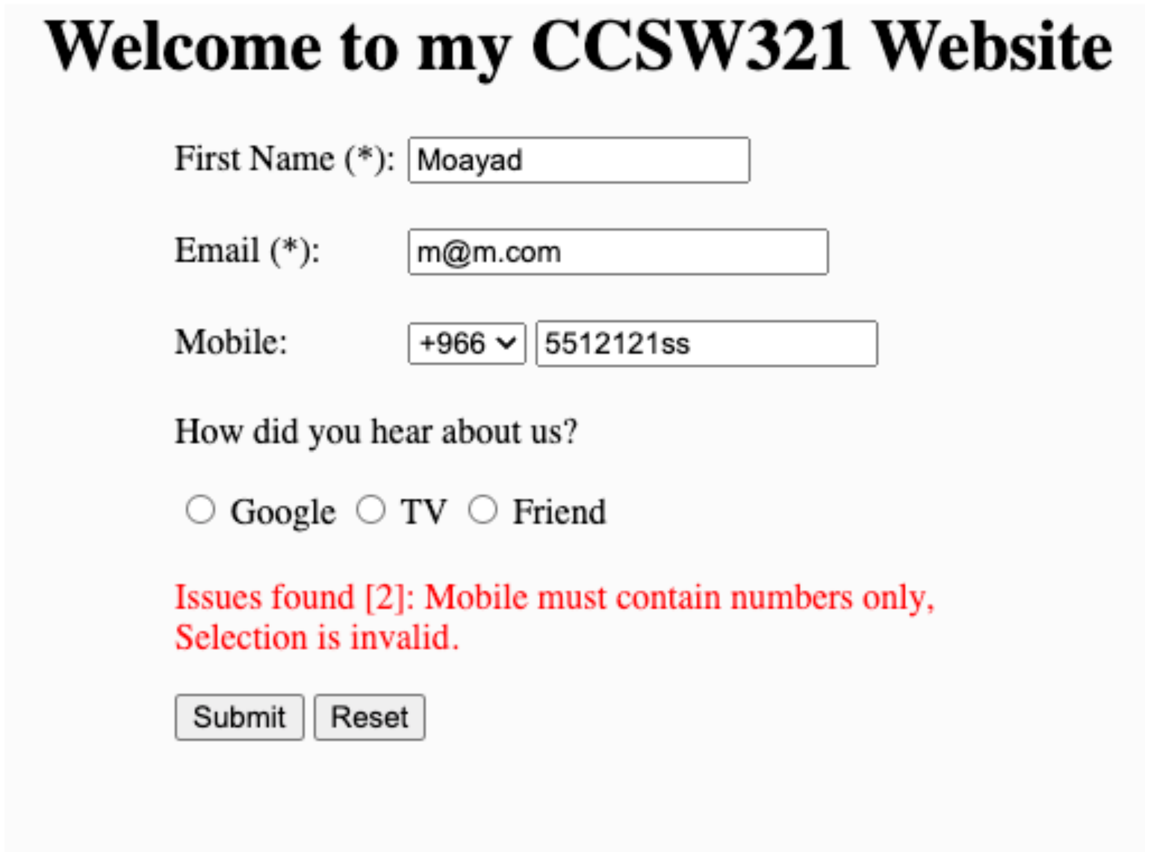
○ Google ○ TV ○ Friend

Issues found [4]: Email format is wrong, Mobile is missing, Mobile must contain numbers only, Selection is invalid.

FORMAT MUST BE "[a-z0-9]+@[a-z]+\.[a-z]{2,4}"

JavaScript Data Validation Layer

- Input = **invalid** mobile
- Output:



Welcome to my CCSW321 Website

First Name (*): Moayad

Email (*): m@m.com

Mobile: +966 5512121ss

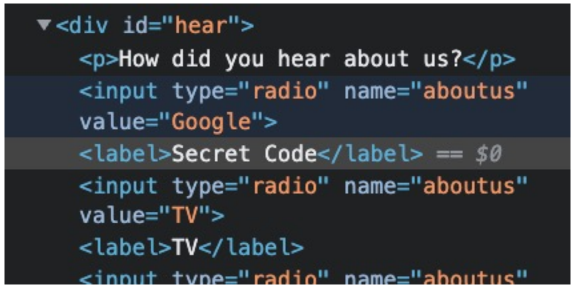
How did you hear about us?

Issues found [2]: Mobile must contain numbers only, Selection is invalid.

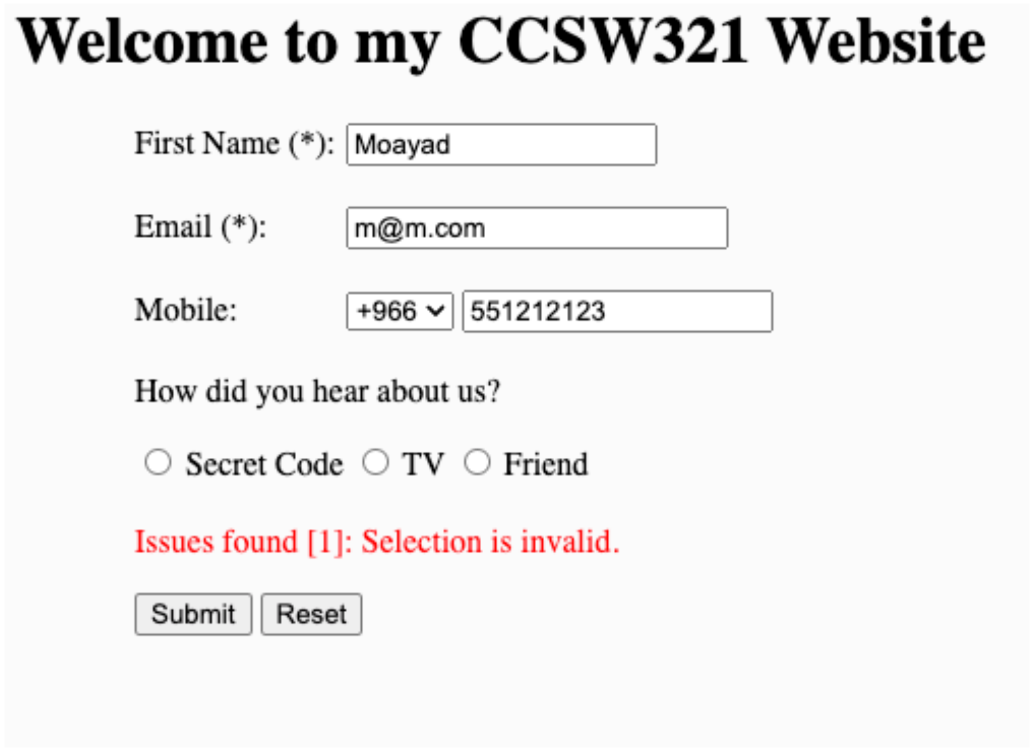
MUST BE NUMBERS ONLY AND 9 DIGITS EXACTLY

JavaScript Data Validation Layer

- Input = **injecting** an **invalid** radio selection
- Output:



```
▼<div id="hear">
    <p>How did you hear about us?</p>
    <input type="radio" name="aboutus"
    value="Google">
    <label>Secret Code</label> == $0
    <input type="radio" name="aboutus"
    value="TV">
    <label>TV</label>
    <input type="radio" name="aboutus"
```



Welcome to my CCSW321 Website

First Name (*):	Moayad
Email (*):	m@m.com
Mobile:	+966 551212123

How did you hear about us?

○ Secret Code ○ TV ○ Friend

Issues found [1]: Selection is invalid.

SELECTION MUST BE FROM WHITELIST

Libraries and Frameworks

- Libraries and frameworks are pre-written code that can help developers build web applications faster and with less effort.
- They provide a set of common functionalities, such as handling HTTP requests, manipulating the DOM, and rendering templates.
- Libraries are smaller and easier to learn than full frameworks.
- They can be added to an existing project gradually, which can help avoid major code rewrites.

Libraries and Frameworks

- Examples of Libraries
  - **jQuery**: A popular JavaScript library that simplifies HTML document traversal and manipulation, event handling, and AJAX.
  - **Moment**.js: A library that provides flexible date and time formatting and parsing.
  - **Lodash**: A utility library that provides a lot of helpful functions for manipulating arrays, objects, and strings.

Libraries and Frameworks

- Examples of Libraries
  - **Angular**: A popular JavaScript framework that provides a full MVC architecture, two-way data binding, and dependency injection.
  - **React**: A JavaScript library that allows developers to build user interfaces in a modular and declarative way, using components.
  - **Vue.**js: A lightweight JavaScript framework that provides reactive data binding and component-based architecture.

Best Practices and Tips

- Maintain clear separation of content, presentation, behavior.
  - HTML with minimal JavaScript inside.
  - Uses DOM to attach and execute all JavaScript functions.
  - Use external .js files for your code, do not mix with HTML.
- Use consistent and clear naming conventions for variables, functions, and other identifiers.
- Avoid global variables and functions, as they can cause naming conflicts and security issues.

Best Practices and Tips

- Make sure to load your .js files in the correct order and after the page is completely loaded.
- Keep your code modular and organized into smaller, reusable functions.
- Use JavaScript libraries and frameworks to simplify complex tasks and improve performance

Any questions?
Please feel free to raise your hands and ask.