

The background of the slide features a complex, stylized graphic. It consists of a network of black lines that resemble a circuit board or a web of connections. These lines are set against a light gray background that contains faint, circular patterns resembling gears or concentric circles. The overall aesthetic is technical and modern.

# JavaScript

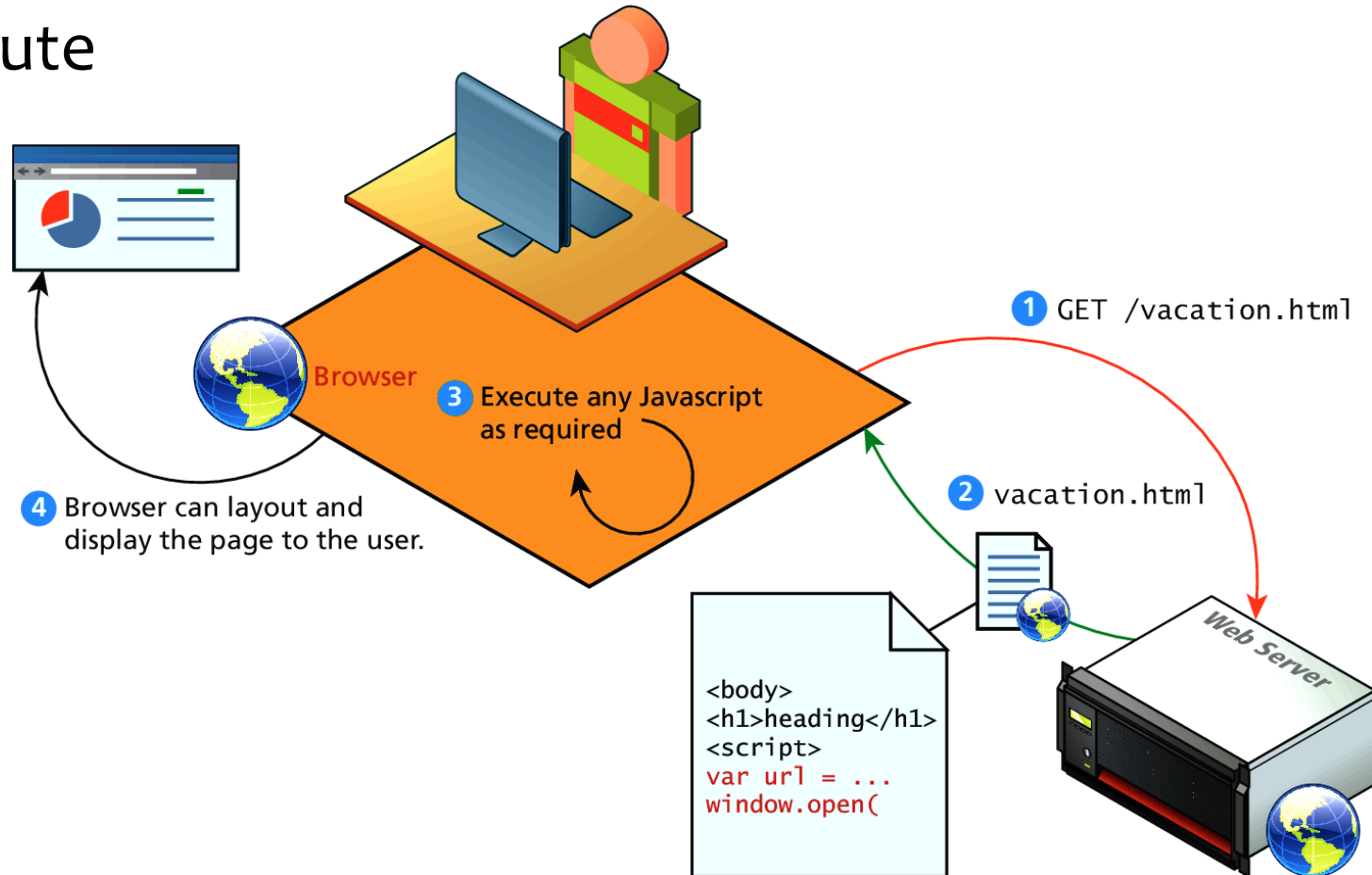
2020/21 COMP3322 Modern Technologies on WWW

# Contents

- JavaScript and its history
- Adding JavaScript
- The JavaScript language
- The Document Object Model (DOM)
- Events
- Form Validation

# Client-Side Scripting

- Let the client compute



# JavaScript isn't related to Java

- Although it contains the word *Java*, JavaScript and Java are vastly **different programming languages** with different uses. Java is a full-fledged compiled, object-oriented language, popular for its ability to run on any platform with a JVM installed.
- Conversely, JavaScript is one of the world's most popular languages, with fewer of the object-oriented features of Java, and runs directly inside the browser, without the need for the JVM.

# What is JavaScript

- JavaScript runs right inside the browser.
- JavaScript is **dynamically typed**.
- JavaScript is object-oriented in that almost everything in the language is an object
  - The objects in JavaScript are **prototype-based** rather than class-based, which means that while JavaScript shares some syntactic features of PHP, Java or C#, it is also quite different from those languages.

# JavaScript History

- JavaScript was introduced by Netscape in their Navigator browser back in 1995.
- JavaScript is in fact an implementation of a standardized scripting language called **ECMAScript**
- In 1998, ECMAScript 2 was released.
- In 1999, ECMAScript 3 was released.
  - Which evolved into what is today's modern JavaScript.
- ECMAScript 5 was released in 2009 and ECMAScript 2015 (the 6<sup>th</sup> edition) was released in 2015.
- At the moment, ECMAScript 2019 (the 10<sup>th</sup> edition) is the latest release.

# Where to place JavaScript?

- JavaScript can be linked to an HTML page in a number of ways.
  - Inline
  - Embedded in the document
  - Link to an external file
- When the browser encounters a block of JavaScript, it generally runs it in order, from top to bottom. This means that you need to be careful what order you put things in.

# Inline JavaScript

- Inline JavaScript refers to add the JavaScript code directly within certain HTML attributes.
- Inline JavaScript is a real maintenance nightmare.

```
<a href="JavaScript:OpenWindow();">more info</a>  
<input type="button" onclick="alert('Are you sure?');">
```



# Embedded JavaScript

- Embedded JavaScript refers to the practice of placing JavaScript code within a `<script>` element.
- You can place any number of scripts in an HTML document.
- Scripts can be placed in the `<body>`, or in the `<head>` section of an HTML page, or in both.

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

# External JavaScript

```
<head>  
  <script src="greeting.js">  
  </script>  
</head>
```

- JavaScript supports this separation by allowing links to an external file that contains the JavaScript.
- By convention, JavaScript external files have the extension .js.
- You can place an external script reference in <head> or <body> as you like.
- The script will behave as if it was located exactly where the <script> tag is located.

# The JavaScript Language

# Basic Syntax

- Semicolon
  - JavaScript generally **does not require semicolons** if you have only one statement on a line.
  - If you place more than one statement on a line, you must separate them with semicolons.
- JavaScript has two ways of writing comments.
  - To write a single-line comment, we can use two slash characters (//).
  - To write a block comment, we write the comment between /\* and \*/ pair.

# Variables

- **Variables** in JavaScript are **dynamically typed**, meaning a variable can be an integer, and then later a string, then later an object, if so desired.
- This simplifies variable declarations as we **do not require** to give the **type** during **declaration**. Instead we use **var** and **let** keywords to declare the variables.

```
var x = 5;  
var y = 6;  
var z;  
let sum = 2;
```

- Variable names are **case-sensitive**.
- The first character of a variable name can be only a-z, A-Z, \$, or \_ (no numbers).
- Suggestions by W3School
  - camelCase is used by JavaScript itself, by jQuery, and other JavaScript libraries.
  - Do not start names with a \$ sign. It will put you in conflict with many JavaScript library names.

# Difference Between var and let

- **let** was introduced in ECMAScript 2015
- It is similar to **var** when is used in functions.
  - They both declare variables that are **local to the function**.
  - A variable declared in a function **cannot be** accessed or visible outside the function.
- However, let behaves differently within a block of code (delimited by { } ) when compare to var.
- As it is related to the scope concept, more explanation and examples will be given when we talk about the Variable Scope.

# Numbers

- All numbers in JavaScript are represented (stored) as **floating-point** values.
- JavaScript uses 64 bits to store a numerical value.
- There are three special values in JavaScript that are considered numbers but don't behave like normal numbers.
  - Infinity, -Infinity, and NaN
    - JavaScript does not raise overflow, underflow, or division by zero errors.
    - A result of computation results in a number larger than the allowed value → Infinity
    - A negative value larger than the allowed negative value → -Infinity
    - Division by zero → Infinity or -Infinity
    - Zero divides zero → NaN as well as Infinity divides infinity → NaN

# Strings

- A string is an **immutable** ordered sequence of 16-bit Unicode characters.
- JavaScript string variables are enclosed within a matched pair of single quotes or double quotes or backticks.
  - 'Single quotes'
  - "Double quotes"
  - `Backticks` - template literal
- You may include a single quote within a double-quoted string or vice versa. But you must escape a quote of the same type by using backslash character.
  - "Say \"Cheer!\""



# Working with Strings

- Use the + operator to concatenate strings

```
> "Hello" + "World";  
◀ "HelloWorld"
```

- Use the length property to get the length of a string

```
> var str = "Working with strings"; str.length;  
◀ 20
```

- There are a number of methods you can invoke on strings, for example:

```
var s = "Hello, world"           // Start with some text.  
s.charAt(0)                     // => "H": the first character.  
s.charAt(s.length-1)           // => "d": the last character.  
s.substring(1,4)                // => "ell": the 2nd, 3rd and 4th characters.  
s.slice(1,4)                    // => "ell": same thing  
s.slice(-3)                     // => "rld": last 3 characters  
s.indexOf("l")                  // => 2: position of first letter l.  
s.lastIndexOf("l")              // => 10: position of last letter l.  
s.indexOf("l", 3)               // => 3: position of first "l" at or after 3
```

# Boolean

- We can also assign a variable a **true** or **false** value.
- JavaScript supports three logical operators on Boolean variables (and expressions):
  - and (&& operator),
  - or (|| operator), and
  - not (! operator).
- Everything in JavaScript has either an inherently “true” or “false” value.
  - null, undefined, 0, and empty strings ("" ) are all inherently false, while every other value is inherently true.

# Comparison Operators

Operator	Description	Matches (x=9)
==	Equals	(x==9) is true (x=="9") is true
===	Exactly equals, including type	(x==="9") is false (x===9) is true
< , >	Less than, Greater Than	(x<5) is false
<= , >=	Less than or equal, greater than or equal	(x<=9) is true
!=	Not equal	(4!=x) is true
!==	Not equal in either value or type	(x!== "9") is true (x!==9) is false

# Null and Undefined

- **null** is a keyword that evaluates to a special value that is usually used to **indicate the absence** of a value.
  - In JavaScript, the data type of null is an object.
- **undefined** represents a variable without a value.
  - It is the value of variables that have not been initialized.
  - The value you get when you query the value of an object property or array element that does not exist.
  - The undefined value is also returned by functions that have no return value, and the value of function parameters for which no argument is supplied.

```
<p>Objects can be emptied by setting the value  
to null</b>.</p>  
<p id="demo"></p>  
<script>  
var person = "Just a name";  
person = null;  
document.getElementById("demo").innerHTML =  
  person + "<br>" + typeof person;  
</script>
```

Objects can be emptied by setting the value to **null**.

```
null  
object
```

```
<p>Variables can be emptied if you set the value  
to undefined</b>.</p>  
<p id="demo"></p>  
<script>  
var car = "Volvo";  
car = undefined;  
document.getElementById("demo").innerHTML =  
  car + "<br>" + typeof car;  
</script>
```

Variables can be emptied if you set the value to **undefined**.

```
undefined  
undefined
```

# Array

- Two ways of creating an array:

```
var primes = [2, 3, 5, 7, 11, 13];  
var names = ["John", "Peter", "Tony"];
```

```
var primes = new Array(2, 3, 5, 7, 11, 13);  
var names = new Array("John", "Peter", "Tony");
```

- JavaScript arrays are **untyped**.
  - Elements in the same array may be of different types.
  - This allows you to create complex data structures, such as arrays of objects and arrays of arrays.
- JavaScript arrays are dynamic.
  - They **grow or shrink** as needed.
- The index of the first element is 0, and the **highest possible index** is  $2^{32} - 2$
- Every array has a length property. It returns the number of array elements.

# Array Methods

Method	Description
<code>concat()</code>	joins two or more arrays, and returns a copy of the joined arrays
<code>join()</code>	joins all elements of an array into a string
<code>push()</code>	adds new elements to the end of an array, and returns the new length
<code>reverse()</code>	reverses the order of the elements in an array
<code>shift()</code>	removes the first element of an array, and returns that element
<code>sort()</code>	sorts the elements of an array
<code>splice()</code>	adds/removes elements from an array
<code>toString()</code>	converts an array to a string, and returns the result
<code>unshift()</code>	adds new elements to the beginning of an array, and returns the new length

# Conditionals

- JavaScript's syntax is **almost identical** to that of PHP, Java, or C when it comes to conditional structures.
- if and if else statements

```
var hourOfDay;    // var to hold hour of day, set it later...
var greeting;    // var to hold the greeting message.
if (hourOfDay > 4 && hourOfDay < 12){
    // if statement with condition
    greeting = "Good Morning";
}
else if (hourOfDay >= 12 && hourOfDay < 20){
    // optional else if
    greeting = "Good Afternoon";
}
else{ // optional else branch
    greeting = "Good Evening";
}
```

- switch statements

```
switch (page)
{
    case "Home":
        document.write("You selected Home")
        break
    case "About":
        document.write("You selected About")
        break
    case "News":
        document.write("You selected News")
        break
    case "Login":
        document.write("You selected Login")
        break
    case "Links":
        document.write("You selected Links")
        break
}
```

# Conditions

- The ternary operator ? :

a    ?    b    :    c

**x = (y==4) ? "y is 4" : "y is not 4";**

Condition

Value  
if true

Value  
if false



# Loops

- for loops

```
var result = 1;
for (var i = 0; i < 10; i++) {
  result = result * 2;
}
console.log(result);    // → 1024
```

- while loops

```
var result = 1;
var i = 0;
while (i < 10) {
  result = result * 2;
  i++;
}
console.log(result);    // → 1024
```

# Loops

- for...of loops
  - The for...of statement creates a loop iterating over iterable objects, e.g., String, Array, array-like objects, Map, Set, etc.

```
var cars = ['BMW', 'Volvo', 'Mini'];  
for (let x of cars) {  
    document.write(x + "<br >");  
}
```

Diagram labels: "element" points to the variable `x` in the loop header. "iterable object" points to the variable `cars` in the loop header.

[https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref\\_state\\_forof](https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_state_forof)

- for...in loops
  - The for...in statement iterates over all enumerable properties of an object that have strings as keys.

```
var person = {fname:"John", lname:"Doe",  
age:25};  
  
var text = "";  
for (let x in person) {  
    text += person[x] + " ";  
}
```

Diagram label: "key" points to the variable `x` in the loop header.

[https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref\\_state\\_forin](https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_state_forin)

# Functions

- When JavaScript is included in a page, the browser **executes** that JavaScript **as soon as it is read** except the code fragment is in a function. It only runs when it is called.
- There are three ways to create a function:
  - Function declaration
  - Function expression
  - Arrow function

# Function Declaration

- They are defined by using the **reserved word function** and then the function name, (optional) parameters, and the body of the function (enclosed in curly braces).

```
function name(parameter1, parameter2, parameter3) {  
    code to be executed  
}
```

- Since JavaScript is dynamically typed, functions do not require a return type, nor do the parameters require type.

```
var x = myFunction(4, 3);    // Function is called, return value will end up in x  
  
function myFunction(a, b) {  
    return a * b;            // Function returns the product of a and b  
}
```

# Function Expression

- A function is created with the keyword function and is **assigned as an expression** to a variable.
- We can create a function without giving it a name (**anonymous function**).
- To call the function, we call the variable name with necessary arguments.

```
var myFunction = function (a, b) { return a*b };  
var x = myFunction(5, 6); // x gets the value 30
```

# Arrow Function

- Instead of using the function keyword, we use an arrow => to declare a function.

- Without argument:

`() => { statements }`      `var HW = () => { console.log("Hello World"); };`

- With one argument:

`arg1 => { statements }`      `var square = a => { return a*a; };`  
`(arg1) => { statements }`

- Multiple arguments:

`(arg1, ..., argn) => { statements }`      `var multiply = (x, y) => { return x * y; };`  
`(arg1, ..., argn) => single_expression`      `var multiply = (x, y) => x * y;`

# Variable Scope

- In JavaScript there are three types of scope:
  - Global scope
  - Function scope
  - Block scope (since ES2015)

# Global Scope

- Variables that are defined outside of any function or block are in GLOBAL Scope.
- All scripts and functions on a web page can access it.
- **Automatically turn** a variable to global
  - If you assign a value to a variable that has not been declared, it will automatically become a **GLOBAL** variable.
  - We can avoid this by setting our script to "**strict mode**".
    - This system responds to this situation by throwing a ReferenceError.
    - Put this statement: 'use strict'; before any other statements in your script or the function.

```
var carName = "Volvo";

// code here can use carName

function myFunction() {

    // code here can also use carName

}
```

```
myFunction();

// code here can use carName

function myFunction() {
    carName = "Volvo";
}
```



# Function (Local) Scope

- Variables declared within a function (by var or let) have the FUNCTION scope.
- Function arguments also work in the FUNCTION scope.
- Variables declared with var or let keywords are quite similar when declared inside a function.

```
var same = 10;

function testFunc (a, b) {
  var same = a; // local "same"
  let hide = b; // local
  console.log("same:", same);
}

console.log("same:", same); //10
testFunc(21, 33);           //21
console.log("hide:", hide); //ReferenceError
```

# Block Scope

- Variables declared with the var keyword can not have Block Scope.
- Variables declared inside a block { } can be accessed from outside the block.

```
{  
    var B = 2;  
}  
// B CAN be accessed here
```

```
var x = 10;  
// Here x is 10  
{  
    var x = 2;  
    // Here x is 2  
}  
// Here x is 2
```

- Variables declared with the let keyword can have Block Scope.
- Variables declared inside a block { } can not be accessed from outside the block:

```
{  
    let B = 2;  
}  
// B can NOT be accessed here
```

```
var x = 10;  
// Here x is 10  
{  
    let x = 2;  
    // Here x is 2  
}  
// Here x is 10
```

# Const

- Constants are **block-scoped**, much like variables defined using the **let** statement.
- The value of a constant can't be changed through reassignment, and it can't be redeclared.

```
var x = 10;  
// Here x is 10  
{  
  const x = 2;  
  // Here x is 2  
}  
// Here x is 10
```

[https://www.w3schools.com/js/tryit.asp?filename=tryjs\\_const](https://www.w3schools.com/js/tryit.asp?filename=tryjs_const)

# Function Closure

- A closure is the combination of a **function** and the **lexical** environment within which that function was declared.
  - This environment consists of any local variables that were in-scope at the time the function closure was created.
- Lexical environment or scope
  - This describes how a parser resolves variable names.
  - Nested functions have access to variables declared in their outer scope.
  - It uses the location where a variable is declared **within the source code** to determine where that variable is available, i.e., within scope.
- Closures are useful because they let you associate some data (the lexical environment) with a function that operates on that data.

# Function Closure

```
> function outer() {  
  var msg1 = "Can you see me?";  
  
  function inner() {  
    var msg2 = "I can see you. "+msg1;  
    return msg2;  
  }  
  
  console.log(msg1);  
  console.log(inner());  
  console.log(msg2);  
}  
  
outer();
```

---

Can you see me?

---

I can see you. Can you see me?

✖ *ReferenceError* { "msg2 is not defined" }

```
> function makeAdder(x) {  
  return function(y) {  
    return x + y;  
  };  
}  
  
var add5 = makeAdder(5);  
var add10 = makeAdder(10);  
  
console.log(add5(2));  
console.log(add10(2));
```

---

7

---

12

# JavaScript Objects

- In JavaScript, **most things are objects**.
  - From core JavaScript features like strings and arrays to the browser APIs built on top of JavaScript.
- **Inside** the object, we find:
  - **A collection of properties**, each of which has **a name and a value**.
    - Property names are strings.
    - The value can be pretty much anything: strings, numbers, arrays, functions, & objects.
  - So we can say that objects map strings to values.
- We group related data and functions (methods) in to a single object, which represents information about the thing we are trying to model, and functionality or behavior that we want it to have.

# Create an Object

- As an object is made up of multiple members, we can create an object by **literally** writing out the object contents.

```
var person = {  
  name: {first: 'Henry', last: 'Lee'},  
  age: 32,  
  gender: 'male',  
  interests: ['music', 'skiing'],  
  greeting: function() {  
    alert('Hi! I\'m ' + this.name.first + '.');  
  }  
};
```

```
var person = {};  
person.name = {first: 'Henry', last: 'Lee'};  
person.age = 32;  
person.gender = 'male';  
person.interests = ['music', 'skiing'];  
person.greeting = function() {  
  alert('Hi! I\'m ' + this.name.first + '.');  
};
```

- An object like this is referred to as **an object literal**.
- Each name/value pair must be **separated by a comma**, and the name and value in each case are separated by a colon.

# What is 'this'?

```
greeting: function() {  
  alert('Hi! I\'m ' + this.name.first + '.');  
}
```

- Within the body of a method, 'this' evaluates to the object on which the method was invoked.
  - In the above example, **this** refers to the person object.
- In the global execution context (outside of any function), **this** refers to the global object, i.e. the window object in the HTML.
- In a function that is not related to an object, **this** refers to the global object (in non-strict mode) or undefined (in strict mode).
  - This applies even when the function is nested inside an object's method.



# Create an Object

- JavaScript uses **special functions** called **constructor** functions to define new objects and their features.
- To create an object, we use the **new** keyword.

```
function Person (first, last, age, gender, interests) {  
  this.name = {  
    'first': first,  
    'last': last  
  };  
  this.age = age;  
  this.gender = gender;  
  this.interests = interests;  
  this.greeting = function() {  
    alert('Hi! I\'m' + this.name.first + '.');  
  };  
}
```

```
var man = new Person('Henry', 'Lee', 32, 'male', ['music', 'skiing']);
```

# Create an Object

- JavaScript has a built-in method called `create()` that allows you to ***create a new object based on any existing object.***
- To create a new object, simply pass the desired prototype object as an argument to the `Object.create()` method.

```
function Person (first, last, age, gender, interests) {  
  :  
  :  
}
```

```
var man = new Person('Henry', 'Lee', 32, 'male', ['music', 'skiing']);
```

```
var boy = Object.create(man);  
console.log(boy.age);      // => 32  
console.log(boy.name.first); // => Henry
```

# Bracket notation

- In previous examples, we used the **dot notation** to access an object's properties and methods.

```
boy.age  
boy.name.first
```

- Another way to access object properties is to use **bracket notation**.

```
boy["age"];  
boy["name"]["first"]
```

- This looks very similar to how we access the items in an array, instead of using an index number to select an item, you are using the name associated with each member's value.
  - It is the reason why objects are sometimes called **associative arrays**.

# Dynamically Add Object Members

- We can add new properties to an object during runtime.

```
man['eyes'] = 'hazel';  
man.farewell = function() { alert("Bye everybody!"); }
```

# The Window Object

- The window object is supported by all browsers. It represents the **browser's window**.
- There can be several window objects at a time, each representing an open browser window.
- Global variables are properties of the window object. Global functions are methods of the window object.
- Even the document object (of the HTML DOM) is a property of the window object.

# The Window Object

- The window object has a number of properties and methods that we can use to interact with it.

Property /method	Description
<code>document</code>	Returns the Document object for the window
<code>history</code>	Returns the History object for the window
<code>location</code>	Returns the Location object for the window
<code>status</code>	Sets or returns the text in the statusbar of a window
<code>alert()</code>	Displays an alert box with a message and an OK button
<code>close()</code>	Closes the current window
<code>focus()</code>	Sets focus to the current window
<code>open()</code>	Opens a new browser window
<code>prompt()</code>	Displays a dialog box that prompts the visitor for input
<code>resizeTo()</code>	Resizes the window to the specified width and height

# try...catch...finally Statement

- The try/catch/finally statement marks a block of code to try and specifies a response should an exception be caught.
  - It handles some or all of the errors that may occur in a block of code
- The **try** statement allows you to define a block of code to be tested for errors while it is being executed.
- The **catch** statement allows you to define a block of code to be executed, if an error occurs in the **try** block.
- The **finally** statement lets you execute code, after try and catch, regardless of the result.

# try...catch...finally Statement

- Syntax

```
try {  
    tryCode - Block of code to try  
}  
catch(err) {  
    catchCode - Block of code to handle errors  
}  
finally {  
    finallyCode - Block of code to be executed  
    regardless of the try / catch result  
}
```

The catch-block specifies an identifier (err in the example) that holds the value of the exception; this value is only available in the scope of the catch-block.

[https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref\\_state\\_finally\\_error](https://www.w3schools.com/jsref/tryit.asp?filename=tryjsref_state_finally_error)



# Import and Export Statements

- JavaScript Import statement is used to **import** bindings that are **exported** by another JavaScript module.
  - To make objects, functions, classes or variables available to the outside world it's as simple as exporting them and then importing them where needed in other files.
- A feature under ES2015 (ES6).

# Exporting

## Default export (only one per module)

- export default ***expression***;
- export default ***function fname***(...) { ... }
- export { nameX as default };

## Named exports (any number)

- export function fname() { ... }
- export var nameJ = ... ;
- export { nameA, nameB, nameC, ... };
- export { nameX as otherX, nameY as otherY, ... };

# Importing

## Import the default of a module

- `import nameX from 'ext.js';`
  - you can give it a name of your choice

## Import all exports of a module

- `import * as extM from 'ext.js';`
  - `extM.fname();`

## Import with alias

- `import { fname as myfunc } from 'ext.js';`

## Import




- `import { nameX, nameY } from 'ext.js';`

# DOM – The Document Object Model

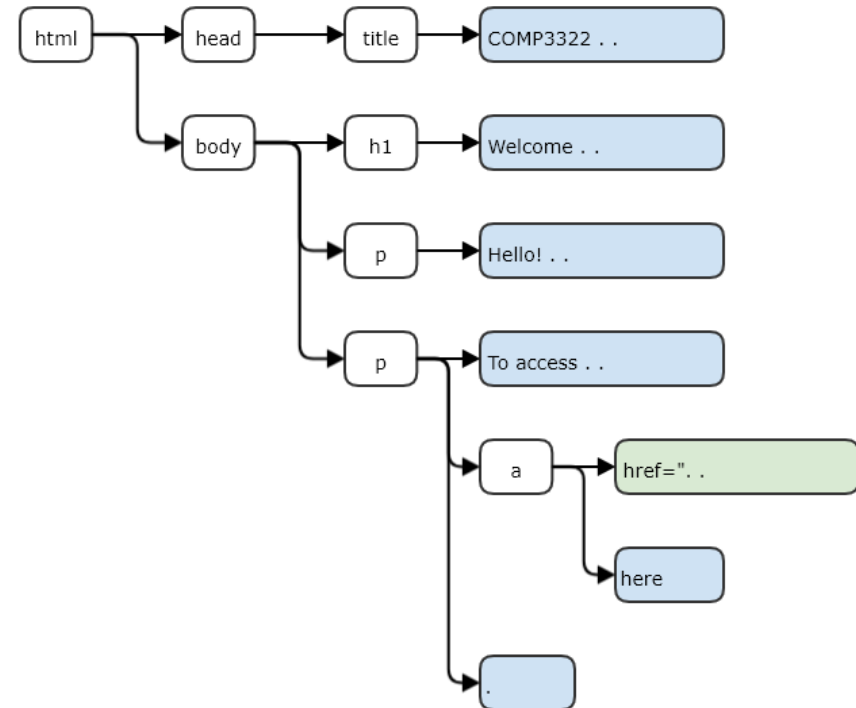
# The DOM

- The Document Object Model (DOM) is a **programming interface** for HTML and XML documents.
- The DOM **represents a web document as nodes and objects**, that allows us to make use of programs to manipulate the content and appearance of the web document.
- According to the W3C, the DOM is a:
  - *Platform- and language-neutral interface that will allow programs and scripts to dynamically access and update the content, structure and style of documents.*

# DOM Nodes

- DOM serves as a map to all the elements on the HTML document.
  - In the DOM, each element within the HTML document is called a node.
  - Nodes are connected, so the whole document can be represented as a tree.
- The DOM is a collection of nodes:
  - element nodes, 
  - text nodes, and 
  - attribute nodes. 

```
<!doctype html>
<html>
  <head>
    <title>COMP3322 Test Page</title>
  </head>
  <body>
    <h1>Welcome Guys!</h1>
    <p>Hello, I am Anthony and this is my course simple home page.</p>
    <p>To access the course's Moodle site, please visit
    <a href=
      "http://moodle.hku.hk/course/view.php?id=71750">here</a>.</p>
  </body>
</html>
```



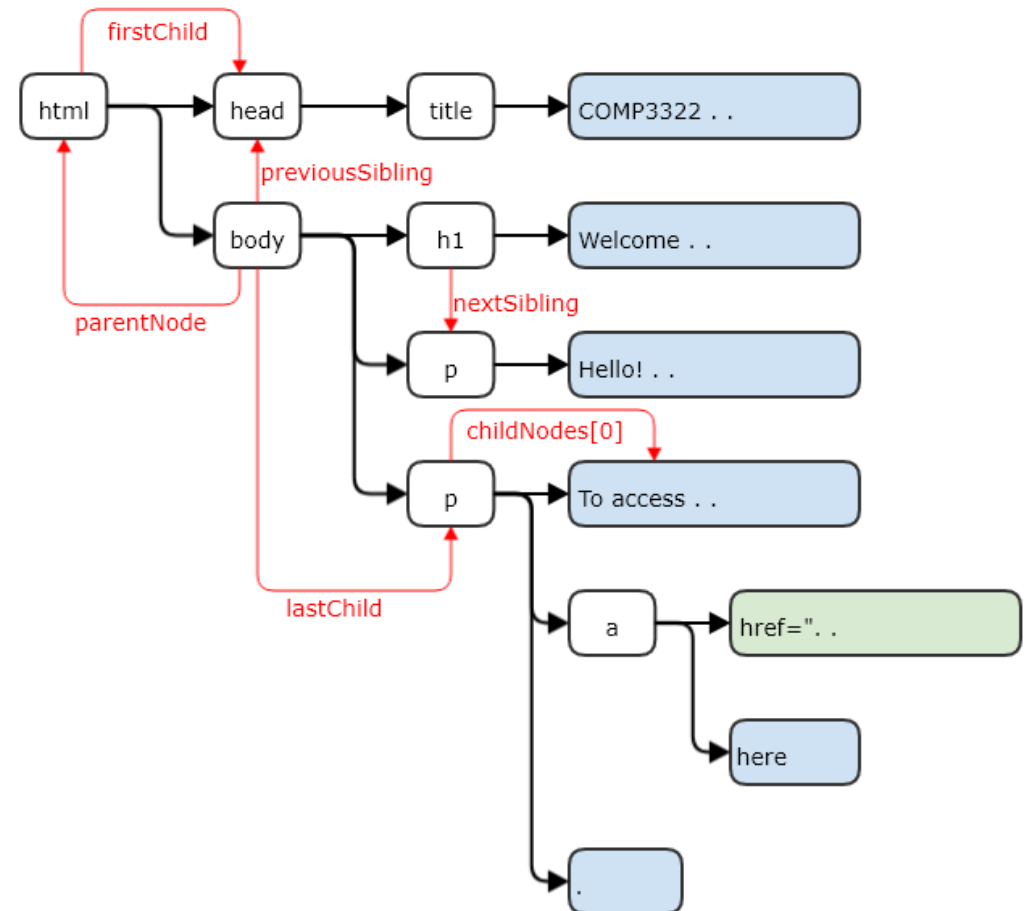
# DOM Nodes

- **All of** the **properties**, **methods**, and **events** available for manipulating and creating web pages are **organized into node objects**.
  - e.g., the document object represents the document, the table object implements the DOM interface for accessing HTML tables, etc.
- Most of the tasks that we typically perform in JavaScript involve finding a node object, and then accessing or modifying it via those properties and methods.

# Basic Node Properties

- Each node also has a number of **basic properties** that you can examine or set.

Property	Description
<code>attributes</code>	Collection of node attributes
<code>childNodes</code>	A NodeList of child nodes for this node
<code>firstChild</code>	First child node of this node.
<code>lastChild</code>	Last child of this node.
<code>nextSibling</code>	Next sibling node for this node.
<code>nodeName</code>	Name of the node
<code>nodeType</code>	Type of the node
<code>nodeValue</code>	Value of the node
<code>parentNode</code>	Parent node for this node.
<code>previousSibling</code>	Previous sibling node for this node.





# Basic Node Methods

- Nodes have a number of methods available.
- Which of these are valid depends on the node's position in the HTML and whether it has parent or child nodes.

Property	Description
<code>appendChild()</code>	Append the new node as the last child.
<code>cloneNode()</code>	Clone a node, and optionally, all of its contents, i.e., include any child nodes of the original node.
<code>hasChildNodes()</code>	Returns a boolean indicating if the element has any child nodes, or not.
<code>insertBefore()</code>	Inserts a child node before the reference node specified in the call.
<code>isSameNode()</code>	Returns a Boolean value indicating whether or not the two nodes are the same.
<code>removeChild()</code>	Removes a child node.
<code>replaceChild()</code>	Replaces one child node of the current one with the second one given in parameter.

# Document Object

- The **document object** is the root JavaScript object **representing the entire HTML document**, and more often it serves as the starting point for our DOM crawling.
- It is the child of the window object. We can use `window.document` to refer to the document object, or simply refer to `document`.
- The **document object** comes with a number of properties and methods for accessing collections of elements.
- Example:

```
console.log(document.title);    // => "COMP3322 Test Page"
```

# Document Object

- Some essential document object properties and methods

Properties / Method	Description
<code>doctype</code>	Returns the Document Type Declaration associated with the document.
<code>forms</code>	Returns a collection of all <form> elements in the document.
<code>head</code>	Returns the <head> element of the document.
<code>title</code>	Sets or gets the title of the current document.
<code>createAttribute()</code>	Creates an attribute node
<code>createElement()</code>	Creates an element node
<code>createTextNode()</code>	Create a text node
<code>getElementById()</code>	Returns the element node whose id attribute matches the parameter.
<code>getElementsByClassName()</code>	Returns a NodeList containing all elements with the specified class name.
<code>getElementsByTagName()</code>	Returns a NodeList containing all elements with the specified tag name.
<code>querySelector()</code>	Returns the first Element node within the document that matches the specified selectors.
<code>querySelectorAll()</code>	Returns all the Element nodes within the document that match the specified selectors.
<code>write()</code>	Writes HTML expressions or JavaScript code to a document.

# Accessing Element Nodes

```
var elm = document.getElementById("header");
```

```
var elm = document.querySelector("#header");
```

```
var elm = document.querySelector("h1");
```

```
<!doctype html>
<html>
  <head>
    <title>COMP3322 Test Page</title>
    <style>
      #header { background-color: yellow;}
    </style>
  </head>
  <body>
    <h1 id="header">Welcome Guys!</h1>
    <p>Hello! I am Anthony and this is my course simple home page.</p>
    <p class="moodle">To access the course's Moodle site, please visit
    <a href="http://moodle.hku.hk/course/view.php?id=77054">here</a>.</p>
  </body>
</html>
```



# Accessing Element Nodes

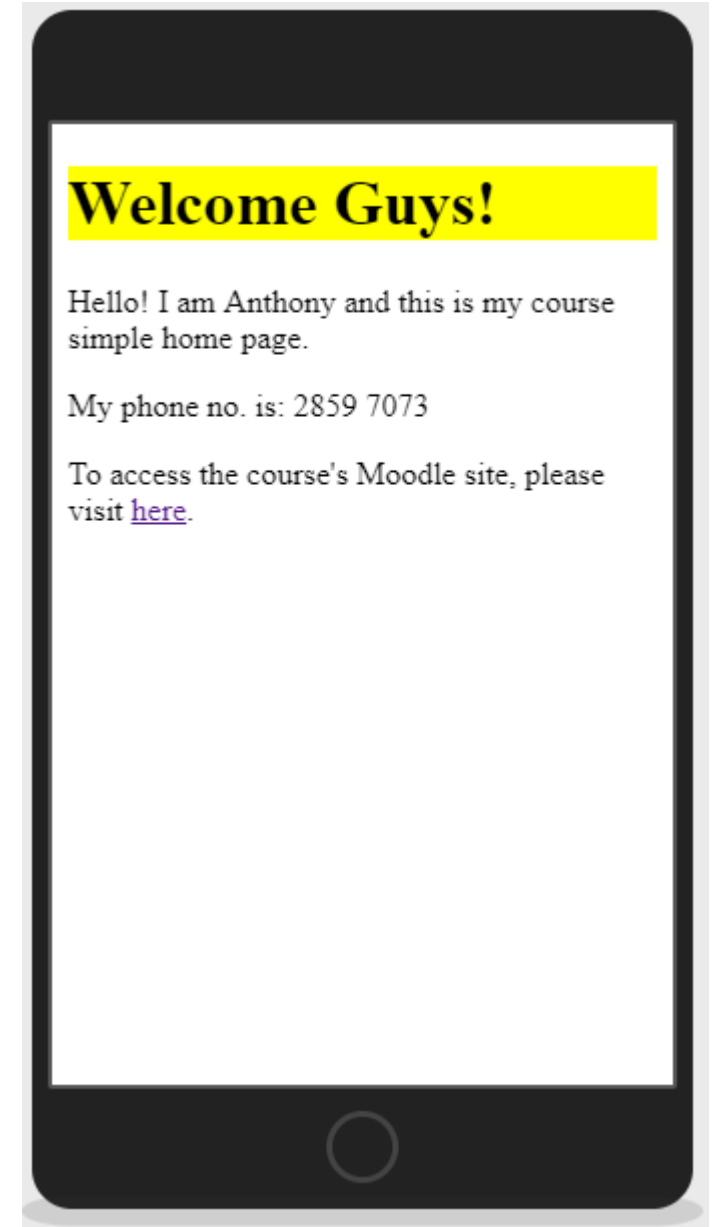
```
<!doctype html>
<html>
  <head>
    <title>COMP3322 Test Page</title>
    <style>
      #header { background-color: yellow;}
    </style>
  </head>
  <body>
    <h1 id="header">Welcome Guys!</h1>
    <p>Hello! I am Anthony and this is my course simple home page.</p>
    <p class="moodle">To access the course's Moodle site, please visit
      <a href="http://moodle.hku.hk/course/view.php?id=77054">here</a>.</p>
  </body>
</html>
```

```
var elm = document.getElementsByTagName('p');
```

```
var elm = document.getElementsByClassName("moodle");
```

# Adding Content

```
<script>
  var para = document.createElement('p');
  var text = document.createTextNode('My phone no. is: 2859 7073');
  para.appendChild(text);
  var refNode = document.getElementsByTagName('p');
  refNode[0].insertAdjacentElement('afterend', para);
</script>
```



# Element node Object

- The type of object returned by the method `document.getElementById()` is an element node object.
  - This represents an HTML element in the hierarchy, contains everything between the opening `<>` and closing `</>` tags for this element.
  - Can itself contain more elements.
- Element objects implement the DOM Element interface, which has its own set of properties and methods.

# Element node Object

- Some essential element object properties and methods

Property / Method	Description
<code>attributes</code>	Returns a collection of this element node's attributes.
<code>className</code>	Sets or returns the value of the class attribute of this element.
<code>id</code>	Sets or returns the value of the id attribute of this element.
<code>innerHTML</code>	Sets or returns the markup content of this element.
<code>style</code>	Sets or returns the value of the style attribute of an element (if is a <code>HTMLElement</code> ).
<code>tagName</code>	Returns the tag name of an element.
<code>addEventListener()</code>	Attaches an event handler to this element.
<code>getAttribute()</code> <code>setAttribute()</code> <code>removeAttribute()</code>	Returns / sets / removes the specified attribute value of this element.



# Accessing Element Attributes

```
<!doctype html>
<html>
  <head>
    <title>COMP3322 Test Page</title>
    <style>
      #header { background-color: yellow;}
    </style>
  </head>
  <body>
    <h1 id="header">Welcome Guys!</h1>
    <p>Hello! I am Anthony and this is my course simple home page.</p>
    <p class="moodle">To access the course's Moodle site, please visit
    <a href="http://moodle.hku.hk/course/view.php?id=77054">here</a>.</p>
  </body>
</html>
```

```
var lnk = document.links[0].href;
```

```
var Alnk = document.getElementsByTagName("a")[0].getAttribute('href');
```

# Adding Content

```
var elm = document.querySelector('p').innerHTML;  
document.querySelector('p').innerHTML =  
    elm + "<p> My phone no. is: 2859 7073 </p>"
```



# Changing Element Styles

```
document.getElementById('header').style.backgroundColor = 'blue';  
document.querySelector('p').style.color = "red";  
document.querySelector('a').style.color = "green";  
document.getElementsByTagName('h1')[0].style.width = "250px";
```



# Event Handling

# Events

- Events are actions or occurrences that happen in the system.
- When system detects an event, it fires a signal of some kind/form and some action can be automatically taken to handle the event.
- In the case of the Web, there are many types of events:
  - The user is clicking the mouse over a certain element.
  - The cursor is hovering over an element.
  - The user is pressing a key on the keyboard.
  - The user is resizing or closing the browser window.
  - A web page is loaded completely.
  - An error occurred.

# Event Handlers

- Each available event has an event handler, which is a block of code (usually a JavaScript function) that will be run when the event fires.

```
window.onclick = function() {  
    /* Any code placed here will run when the user  
       clicks anything within the browser window */  
};
```

- When such a block of code is defined to be run in response to an event firing, we say we are registering an event handler.

# Common Web Events

- Mouse events

Attribute	Description
onclick	The mouse clicks an element.
ondblclick	Double-click on an element.
onmousedown	A mouse button is pressed down on an element.
onmousemove	The mouse pointer is moving while it is over an element.
onmouseout	The mouse pointer moves out of an element.
onmouseover	The mouse pointer moves over an element.
onmouseup	A mouse button is released over an element.

# Common Web Events

- Window events

Attribute	Description
onerror	An error occurs when the document or an image loads.
onload	A page (including all images) is finished loading.
onresize	Fires when the browser window is resized.
onunload	When another page is loaded or when the browser window is closed.

- Keyboard events

Attribute	Description
onkeypress	When a user presses a key. (Deprecated)
onkeydown	When a user is pressing a key.
onkeyup	When a user releases a key.

To get more information on HTML events, please visit: [https://www.w3schools.com/tags/ref\\_eventattributes.asp](https://www.w3schools.com/tags/ref_eventattributes.asp)

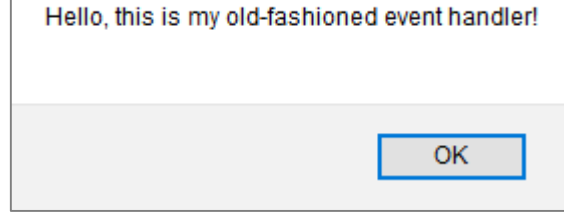


# Registering Event Handlers

- There are three common methods for applying event handlers to elements within web pages:
  - As an HTML attribute
  - As a method attached to the element
  - Using `addEventListener()`

## As an HTML attribute

Press me



- You can specify the function to be run in an attribute in the markup.

```
<button onclick="alert('Hello, this is my old-fashioned event handler!');">  
  Press me</button>
```

- They are considered bad practice.
  - It is not a good idea to mix up your HTML and your JavaScript, as it becomes hard to parse.
  - Could lead to a maintenance nightmare.

<https://i.cs.hku.hk/~atctam/c3322/JS/Event-demo1.html>

## As a Method

- We can keep the code strictly within the `<script>`.

```
<button id="btn">Press me</button>
<script>
  var btn = document.getElementById("btn");
  btn.onclick = function() {
    alert('Hello, this is my old-fashioned event handler!');
  };
</script>
```

- This approach has the benefit of simplicity and ease of maintenance, but has a major drawback.
  - We can bind only **one event handler to an item** at a time with this method.

## Using addEventListener()

```
<button id="btn">Press me</button>
<script>
  var btn = document.getElementById("btn");
  function btnClick () {
    alert('Hello, this is my old-fashioned event handler!');
  };
  btn.addEventListener('click', btnClick);
</script>
```

- Inside the addEventListener() function, we specify two parameters:
  - the name of the event, e.g, click
  - the code of the handler function
- This approach allows us to keep the logic within the scripts and allows us to **perform multiple bindings** on a single object.
  - Both handlers would run when the event occurs.

## removeEventListener()

- We can remove event handler code using removeEventListener()
- Same as addEventListener(), we specify two parameters:
  - the name of the event
  - the handler function

# Event Object

- No matter which type of event we encounter, they are just another type of **DOM objects** and the event handlers associated with them can **access and manipulate** them.
- Typically we see the events passed to the function handler as a parameter named *e*, *evt*, or *event*.

```
function someHandler(e) {  
    // e is the event object, which represents the event that triggered this handler.  
}
```

# Event Propagation – Bubbling

- When an event is fired on an element:
  - The browser first checks whether that element has an event handler on that event, and runs it if so.
  - Afterwards, it checks the next immediate ancestor element and does the same thing, then the next ancestor, and so on until it reaches the `<html>` element.
- In that case, one user action can trigger multiple event handlers in action.
- This is the default behavior of modern browsers.
- The exact opposite behavior is called Capturing.

## Event Propagation – Capturing

- The browser checks to see if the **onclicked** element's outer-most ancestor (<html>) has an onclick event handler registered for capturing, and runs it if so.
- Then it moves on to the next element inside <html> and does the same thing, then the next one, and so on until it reaches the **onclicked** element.
- To use the Capturing behavior, we set the third argument of the `addEventListener()` to `true`, which is set to `false` by default.

```
element.addEventListener(event, function, useCapture)
```

```
addEventListener('click', bgChange, true)
```



# Event Object

- Some essential properties and methods of the event object

Property / Method	Description
<b>bubbles</b>	Indicates whether the event bubbles property is enable.
<b>cancelable</b>	Returns a Boolean value indicating whether or not an event is a cancelable event. The event is cancelable if it is possible to prevent the events default action to be happened.
<b>currentTarget</b>	Refers to the element to which the current event handler is attached to, as opposed to target which identifies the element on which the event occurred.
<b>defaultPrevented</b>	Returns whether or not the preventDefault() method was called for the event.
<b>target</b>	Returns the element that triggered the event.
<b>type</b>	Returns the name of the event.
<b>preventDefault()</b>	Instructs the system to cancel the default action if the event is cancelable.
<b>stopPropagation()</b>	Prevents further propagation of an event during event propagation.

# Form Validation

# Validating Forms – Client-Side

- **Client-side validation** is validation that occurs in the browser, before the data has been submitted to the server.
  - Doing that on the client side will reduce the number of incorrect submissions, thereby reducing server load.
- There are a number of common validation activities including email validation, number validation, and data validation.
- With HTML5, form validation becomes much easy.
  - This generally **does not require** JavaScript.
  - Most modern browsers support this feature.
  - However, the behavior is more or less depended on the browsers.
- Using JavaScript, we can have more control and **customization** of the validation in the client-side.

# HTML5 Form elements (Recap)

Text

Username:

Email

Please enter your email address:

Textarea

URL

Please enter your website address:

Password

Password:

Please enter a URL.

Checkbox

Please select your favorite music service(s):

☒ Apple Music ☐ Amazon Music ☒ Spotify

Date

Departure date:

Select

What smart device do you use for studying?

August 2018

Sun	Mon	Tue	Wed	Thu	Fri	Sat
29	30	31	1	2	3	4
5	6	7	8	9	10	11
12	13	14	15	16	17	18
19	20	21	22	23	24	25
26	27	28	29	30	31	1

# HTML Built-in Validation

- Most of the input are text contents, numbers, strings with special patterns, is required or not, etc.
- HTML uses **validation attributes** on form elements to validate most user input.
  - With the attributes, we can specify rules such as:
    - A required field is still empty. **required**
    - Exceed the maximum length. **minlength** **maxlength**
    - The number is too large or small. **min** **max**
    - The input is not a valid email address or URL.
    - The input does not match the required pattern. **pattern**

# Example

## HTML

```
<h1>Account Registration Form</h1>
<form id="RegForm">
  <p>
    <label for="name">Student Name:</label>
    <input type="text" id="name" name="name" maxlength="50" required>
  </p>
  <p>
    <label for="number">Student No.:</label>
    <input type="text" id="number" name="number" maxlength="10" pattern="[0-9]{6}" required>
  </p>
  <p>
    <label for="age">Age:</label>
    <input type="number" id="age" name="age" min="17" max="30" step="1" pattern="[0-9]+">
  </p>
  <p>
    <label for="email">Student Email:</label>
    <input type="email" id="email" name="email" required>
  </p>
  <input class="btn" type="submit" value="Submit">
  <input class="btn" type="reset">
</form>
```

Specifies the maximum number of characters allowed

The min & max attributes specify the minimum and maximum value for an input

Specifies that an input field must be filled out before submitting the form

## Account Registration Form

Student Name:

Student No.:

Age:

Student Email:

# Example

## HTML

```
<h1>Account Registration Form</h1>
<form id="RegForm">
  <p>
    <label for="name">Student Name:</label>
    <input type="text" id="name" name="name" maxlength="50" required>
  </p>
  <p>
    <label for="number">Student No.:</label>
    <input type="text" id="number" name="number" maxlength="10" pattern="3015[0-9]{6}" required>
  </p>
  <p>
    <label for="age">Age:</label>
    <input type="number" id="age" name="age" min="17" max="30" step="1" pattern="[0-9]+">
  </p>
  <p>
    <label for="email">Student Email:</label>
    <input type="email" id="email" name="email" required>
  </p>
  <input class="btn" type="submit" value="Submit">
  <input class="btn" type="reset">
</form>
```

With 10 numerical characters  
starting with '3015'

Must be a number  
with 1 or more digit

## Account Registration Form

Student Name:

Student No.:

Age:

Student Email:

Submit

Reset

# HTML Built-in Validation

- If the input data doesn't satisfy the prescribed rule, it is considered **invalid**; otherwise, it is considered **valid**.
- With the valid / invalid status, we can make use of the built-in CSS **pseudo-class** for the visualization.
- When an element XX is valid:
  - This matches the XX:valid styling rule, the system will apply the specific style to that valid element XX.
- When an element XX is invalid:
  - This matches the XX:invalid styling rule and the specific style will be applied.
  - **The browser will block the form from submission and display an error message.**



# Example

CSS

```
<style>
  label {
    display: inline-block;
    width: 110px;
  }
  .btn{
    display: inline-block;
    width: 80px;
  }
  input:invalid {
    border: 1px dashed red;
  }
  input:valid {
    border: 2px solid black;
  }
</style>
```

If the input element is in invalid status, sets the border to dashed red color

If the input element is in valid status, sets the border to black color

## Account Registration Form

Student Name:

Student No.:

Age:

Student Email:

<https://i.cs.hku.hk/~atctam/c3322/JS/demo-form-val1.html>

# Customization

- HTML5 provides another API called **constraint validation**.
- The purpose is for us to **check the state** of a form element; given the state, we can take appropriate actions.
- It's also possible to **change the text** of the error message.

# Constraint Validation API

Property / Method	Description
<code>validity</code>	A <code>ValidityState</code> object describing the validity state of the element.
<code>validity.patternMismatch</code>	Set to true, if an element's value does not match its pattern attribute.
<code>validity.rangeOverflow</code>	Set to true, if an element's value is greater than its max attribute.
<code>validity.rangeUnderflow</code>	Set to true, if an element's value is less than its min attribute.
<code>validity.tooLong</code>	Set to true, if an element's value exceeds its maxLength attribute.
<code>validity.valueMissing</code>	Set to true, if an element (with a required attribute) has no value.
<code>checkValidity()</code>	Returns true if an input element contains valid data.
<code>setCustomValidity()</code>	Adds a custom error message to the element

# Example

## JavaScript

```
<script>
var email = document.getElementById("email");
email.addEventListener("input", function (event) {
    if (email.validity.typeMismatch) {
        console.log(email.validationMessage);
        email.setCustomValidity("Enter a valid email address");
    } else {
        email.setCustomValidity("");
    }
});

var snum = document.getElementById("number");
snum.addEventListener("input", function (event) {
    if (snum.validity.patternMismatch) {
        console.log(snum.validationMessage);
        snum.setCustomValidity("Must be 10 digits starts with 3015");
    } else {
        snum.setCustomValidity("");
    }
});
</script>
```

When user enters something to the input element

This function checks whether the value matches the specific pattern

## Account Registration Form

Student Name:

Student No.:

Age:

Student Email:

## Account Registration Form

Student Name:

Student No.:

Age:

Student Email:

Must be 10 digits starts with 3015

# Example

## JavaScript

```
<script>
var email = document.getElementById("email");
email.addEventListener("input", function (event) {
    if (email.validity.typeMismatch) {
        console.log(email.validationMessage);
        email.setCustomValidity("Enter a valid email address");
    } else {
        email.setCustomValidity("");
    }
});

var snum = document.getElementById("number");
snum.addEventListener("input", function (event) {
    if (snum.validity.patternMismatch) {
        console.log(snum.validationMessage);
        snum.setCustomValidity("Must be 10 digits starts with 3015");
    } else {
        snum.setCustomValidity("");
    }
});
</script>
```

## Account Registration Form

Student Name:

Student No.:

Age:

Student Email:

## Account Registration Form

Student Name:

Student No.:

Age:

Student Email:

# Readings

- MDN Web Docs
  - JavaScript First Steps
    - [https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First\\_steps](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/First_steps)
  - JavaScript Building Blocks
    - [https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building\\_blocks](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Building_blocks)
  - Manipulating Documents
    - [https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side\\_web\\_APIs/Manipulating\\_documents](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Manipulating_documents)

# References

- Some slides are borrowed from the book:
  - Fundamentals of Web Development by Randy Connolly and Ricardo Hoar, published by Pearson.
- W3Schools.com
  - JavaScript Tutorial
    - <https://www.w3schools.com/js/default.asp>
- JavaScript Info
  - <https://javascript.info/>