

CSCB20

Introduction to Databases and Web Application

Week 7 - Media Queries and JavaScript

Dr. Purva R Gawde

Topics

- Media Queries

- Basics
- Media Feature Rules
- Logical operators
- Choosing breakpoints
- Example

- Javascript

- Introduction
- Grammar of JS
- Logic and control
- Events
- Examples

Media Queries

- Allows to create **responsive websites** across all screen sizes, ranging from desktop to mobile.
- Media queries are used for the following:
 - To conditionally apply styles with the CSS `@media` and `@import` at-rules.
 - To target specific media for the `<style>`, `<link>`, `<source>`, and other HTML elements with the `media=` attribute.
 - To test and monitor media states using the `Window.matchMedia()` and `MediaQueryList.addListener()` JavaScript methods.

Media Query Basics

```
@media media-type and (media-feature-rule) {  
    /* CSS rules go here */  
}
```

- media-type : what kind of media this code is for (e.g. print, or screen).
 - all
 - print
 - screen
- media expression: test that must be passed for the contained CSS to be applied.

```
@media print {  
    body {  
        font-size: 12pt;  
    }  
}
```

Media Feature Rules

```
@media screen and (width = 600px) {  
  body {  
    color: red;  
  }  
}
```

- The width (and height) media features can be used as ranges

```
@media screen and (max-width = 600px) {  
  body {  
    color: red;  
  }  
}
```

- Orientation:

```
@media (orientation: landscape) {  
  body {  
    color: rebeccapurple;  
  }  
}
```

Media Feature Rules contd..

- Pointing devices:

```
@media (hover: hover) {  
  body {  
    color: purple;  
  }  
}
```

- Pointer media

```
@media (pointer: fine) {  
  body {  
    color: purple;  
  }  
}
```

Logical Operators in Media Queries

- **And**


```
@media screen and (min-width: 600px) and (orientation: landscape) {  
  body {  
    color: blue;  
  }  
}
```

- **Or**

```
@media screen and (min-width: 600px), screen and (orientation: landscape) {  
  body {  
    color: blue;  
  }  
}
```

- **Not**

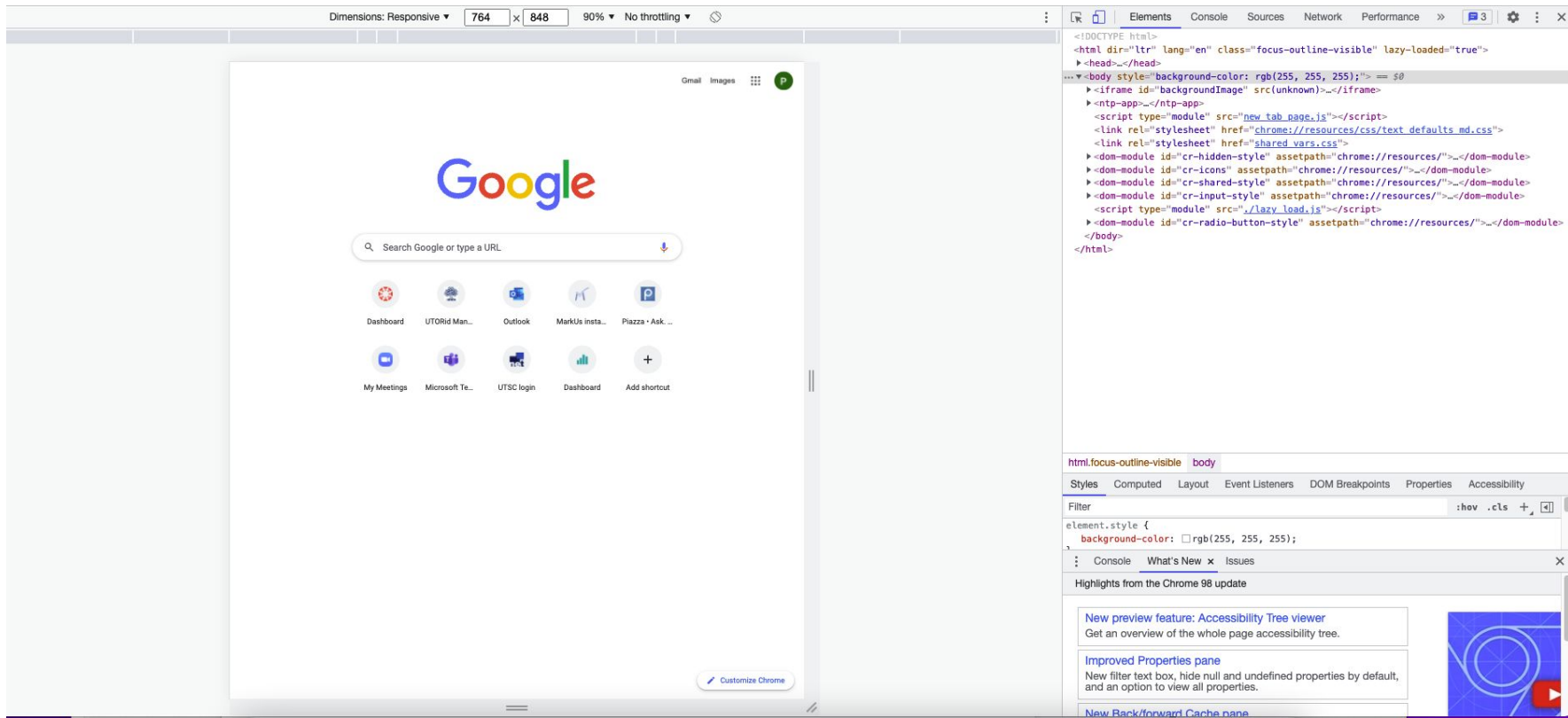
```
@media not all and (orientation: landscape) {  
  body {  
    color: blue;  
  }  
}
```



Examples of Media Queries discussed before MediaQueryEx1

How to choose breakpoints

- The Responsive Design Mode



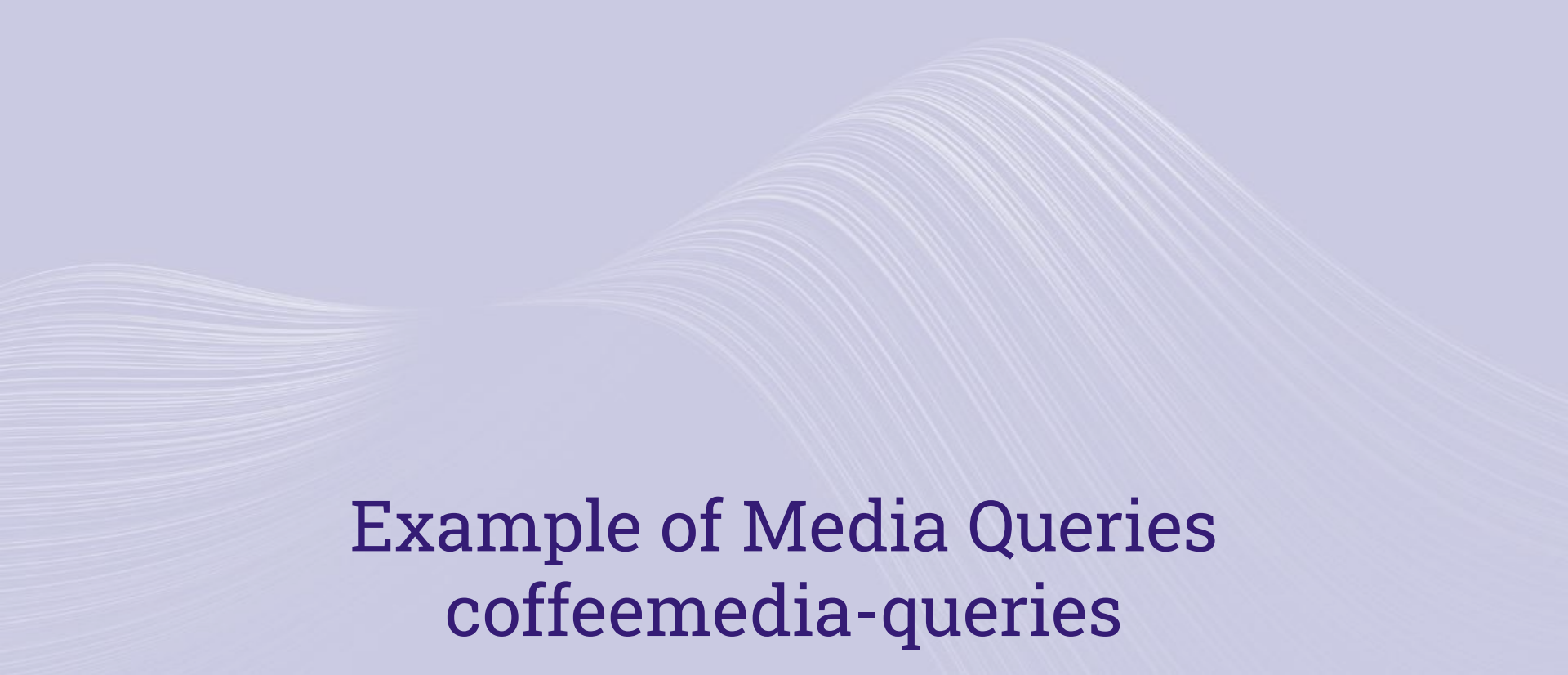
How to begin??

- Two options:

- Widest view

OR

- Mobile first design



Example of Media Queries

coffeemedia-queries

CSS selectors we have discussed so far:

Basic Selectors:

Universal selector

Selects all elements. Optionally, it may be restricted to a specific namespace or to all namespaces.

Syntax: `* ns|* *|*`

Example: `*` will match all the elements of the document.

Type selector

Selects all elements that have the given node name.

Syntax: `elementname`

Example: `input` will match any `<input>` element.

Class selector

Selects all elements that have the given class attribute.

Syntax: `.classname`

Example: `.index` will match any element that has a class of "index".

ID selector

Selects an element based on the value of its id attribute. There should be only one element with a given ID in a document.

Syntax: `#idname`

Example: `#toc` will match the element that has the ID "toc".

Attribute selector

Selects all elements that have the given attribute.

Syntax: `[attr] [attr=value] [attr~=value] [attr|=value] [attr^=value] [attr$=value] [attr*=value]`

Example: `[autoplay]` will match all elements that have the autoplay attribute set (to any value).

CSS selectors we have discussed so far:

Grouping selectors

Selector list

The `,` selector is a grouping method that selects all the matching nodes.

Syntax: A, B Example: `div, span` will match both `` and `<div>` elements.

Combinators

Descendant combinator

The `" "` (space) combinator selects nodes that are descendants of the first element.

Syntax: A B

Example: `div span` will match all `` elements that are inside a `<div>` element.

Child combinator

The `>` combinator selects nodes that are direct children of the first element.

Syntax: A > B

Example: `ul > li` will match all `` elements that are nested directly inside a `` element.

General sibling combinator

The `~` combinator selects siblings. This means that the second element follows the first (though not necessarily immediately), and both share the same parent.

Syntax: A ~ B

Example: `p ~ span` will match all `` elements that follow a `<p>`, immediately or not.

Adjacent sibling combinator

The `+` combinator matches the second element only if it immediately follows the first element.

Syntax: A + B

Example: `h2 + p` will match all `<p>` elements that immediately follows `<h2>` element.

Column combinator

The `||` combinator selects nodes which belong to a column.

Syntax: A || B

Example: `col || td` will match all `<td>` elements that belong to the scope of the `<col>`.

CSS selectors (Pseudo)

Pseudo classes

The : pseudo allow the selection of elements based on state information that is not contained in the document tree.

Example: `a:visited` will match all `<a>` elements that have been visited by the user.

Pseudo elements

The :: pseudo represent entities that are not included in HTML.

Example: `p::first-line` will match the first line of all `<p>` elements.



JavaScript

JavaScript Introduction

- What is JavaScript?
 - Javascript is a scripting language that was introduced to make web pages alive and be interactive with the user.
- What is ES?
 - ECMA (European Computer Manufacturers Association) International carved out standard specification called ECMAScript (ES) which all browser vendors could implement.
 - The 12th edition, or ECMAScript 2021, was published in June 2021

Why Learn JavaScript?

- A high level programming language used to make web pages interactive
 - Supercharges HTML with animation, interactivity, and dynamic visual effects.
 - Web pages respond instantly to actions like clicking a link, filling out a form
- Scripting language
 - Uses interpreter
- Runs on client's browser using JavaScript engine:
 - Chrome: v8
 - Firefox: SpiderMonkey
- Object based language
- JavaScript is everywhere - frontend and backend (node.js)

jQuery

- jQuery is a JavaScript library intended to make JavaScript programming easier and more fun.
- simplifies difficult tasks and solves cross-browser problems

Adding JavaScript to a Page

- Using a `<script></script>` tag
- In the HTML file

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <title>My Web Page</title>
  <script>
    alert('hello world!');
  </script>
</head>
```

- External file with .js extension

```
<!doctype html>
<html>
<head>
  <meta charset="UTF-8">
  <title>My Web Page</title>
  <script src="navigation.js"></script>
</head>
```



Grammar of JavaScript

Statements, built-in functions, types of data, variables

Statements and Built-in functions

- Statement
 - A JavaScript statement is a basic programming unit, usually representing a single step in a JavaScript program.
 - Example: `alert('Hello World!');`
- Built-in functions
 - Works like commands to get things done
 - `alert()` makes the web browser open a dialog box and display a message
 - Example: `alert()`, `document.write()`, `isNaN()`

Variables

- Three options:
 - `var`: Declares a variable (local and global variables, depending on the execution context), optionally initializing it to a value.
 - `let`: Declares a block-scoped, local variable, optionally initializing it to a value.
 - `const` : Declares a block-scoped, read-only named constant.
- Variable Naming conventions:
 - Variable names must begin with a letter, \$, or _
 - Variable names can only contain letters, numbers, \$, and _
 - Variable names are case-sensitive
 - Avoid keywords

Variable Scope

- Global scope: When you declare a variable outside of any function, it is called a global variable

```
if (true) {  
  var x = 5;  
}  
console.log(x); // x is 5
```

- Block statement scope

```
if (true) {  
  let y = 5;  
}  
console.log(y); // ReferenceError: y is not defined
```

Evaluating Variables

```
var a;  
console.log('The value of a is ' + a); // The value of a is undefined
```

```
console.log('The value of b is ' + b); // The value of b is undefined  
var b;  
// ??? What is happening here???
```

```
console.log('The value of c is ' + c); // Uncaught ReferenceError: c is not defined
```

```
let x;  
console.log('The value of x is ' + x); // The value of x is undefined
```

```
console.log('The value of y is ' + y); // Uncaught ReferenceError: y is not defined  
let y;
```


Variable Hoisting

- Variables are hoisted to the top of function or statement
- Variables that are hoisted return a value of **undefined**

```
console.log(x === undefined); // true  
var x = 3;
```

Is equivalent to

```
var x;  
console.log(x === undefined); // true  
x = 3;
```

- In ECMAScript 2015, **let** and **const** are hoisted but not initialized.

```
console.log(x); // ReferenceError  
let x = 3;
```

Because of hoisting, all *var* statements in a function should be placed as near to the top of the function as possible. This best practice increases the clarity of the code.



Image Taken from Wikipedia: CC John E. Thwaites, Public domain, via Wikimedia Commons

Types of Data

- **Boolean.** true and false.
- **null.** A special keyword denoting a null value. (Because JavaScript is case-sensitive, null is not the same as Null, NULL, or any other variant.)
- **undefined.** A top-level property whose value is not defined.
- **Number.** An integer or floating point number.
- **BigInt.** An integer with arbitrary precision.
- **String.** A sequence of characters that represent a text value. For example: "Howdy"
- **Symbol** (new in ECMAScript 2015). A data type whose instances are unique and immutable.
- and **Object**

JavaScript is dynamically typed language

```
var num = 5;  
num = 'not num';
```

Allowed in JS

Working with Data types and Variables

- Combining strings

```
var firstName = 'John';  
var lastName = 'Smith';  
var fullName = firstName + lastName;
```

- Combining Numbers and Strings

```
var numOfVisits = 101;  
var message = 'You have visited this site ' + numOfVisits + ' times.';
```

```
var numOfShoes = '2';  
var numOfSocks = 4;  
var totalItems = Number(numOfShoes) + numOfSocks;
```

Changing values in Variables

```
var score = 0;  
score = score + 100;  
  
let score1 = 0;  
score1++;  
  
const score2 = 0;  
score2++;
```

Operator	What it does	How to use it	The same as
+=	Adds value on the right side of equal sign to the variable on the left.	score += 10	score = score + 10
-=	Subtracts value on the right side of the equal sign from the variable on the left.	score -= 10	score = score - 10
*=	Multiplies the variable on the left side of the equal sign and the value on the right side of the equal sign.	score *= 10	score = score * 10
/=	Divides the value in the variable by the value on the right side of the equal sign.	score /= 10	score = score / 10
++	Placed directly after a variable name, ++ adds 1 to the variable.	score++	score = score + 1
--	Placed directly after a variable name, -- subtracts 1 from the variable.	score--	score = score - 1

Arrays

- Declaring an array

```
var days = ['Mon', 'Tues', 'Wed', 'Thurs', 'Fri', 'Sat', 'Sun'];
```

OR

```
var days = new Array('Mon', 'Tues', 'Wed');
```

- You can store any mix of values in an array.

- ```
var prefs = [1, 223, 'https://www.utoronto.ca/home/', false];
```

- Accessing elements: 

```
prefs[2]
```

- Adding an element: 

```
prefs.push('Hello World');
```

- Deleting the last element: 

```
prefs.pop();
```

- Checking for array:

- ```
console.log(Array.isArray(prefs));
```

- Finding the index

- ```
console.log(prefs.indexOf(223));
```

# More on Arrays

| Method           | Original array       | Example code   | Resulting array | Explanation                                          |
|------------------|----------------------|----------------|-----------------|------------------------------------------------------|
| .length property | var p =<br>[0,1,2,3] | p[p.length]=4  | [0,1,2,3,4]     | Adds one value to the end of an array.               |
| push()           | var p =<br>[0,1,2,3] | p.push(4,5,6)  | [0,1,2,3,4,5,6] | Adds one or more items to the end of an array.       |
| unshift()        | var p =<br>[0,1,2,3] | p.unshift(4,5) | [4,5,0,1,2,3]   | Adds one or more items to the beginning of an array. |
| pop()            | var p =<br>[0,1,2,3] | p.pop()        | [0,1,2]         | Removes the last item from the array.                |
| shift()          | var p =<br>[0,1,2,3] | p.shift()      | [1,2,3]         | Removes the first item from the array.               |

# Objects

---

- You can conceptualize many of the elements of the JavaScript language, as well as elements of a web page, as **objects**.
- Example of Objects in JavaScript:
  - a browser window, a document, a string, a number, and a date

| Object                  | Property | Method    |
|-------------------------|----------|-----------|
| document                | title    |           |
|                         | url      |           |
| ['Kate','Graham','Sam'] | length   | write()   |
|                         |          | push()    |
|                         |          | pop()     |
|                         |          | unshift() |

# Objects

---

- **Defining an object:**

```
const student = {
 firstName: 'John',
 lastname: 'Doe',
 age: 30,
 courses: ['A48', 'B20', 'C43'],
 address: {
 street: 'Main st',
 city: 'Toronto',
 province: 'Ontario'
 }
}
```

- **Accessing an object:**

```
console.log(student.firstName);
student.email = 'student@mail.utoronto';
```



# Arrays of Objects

---

```
const students = [
 {
 id: 1,
 classes: 'CSCB20',
 grade: 78
 },
 {
 id: 2,
 classes: 'CSCA48',
 grade: 90
 },
 {
 id: 3,
 classes: 'CSCA08',
 grade: 91
 }
];
```



# Adding Logic and control

Conditionals, loops, functions

# Comparison operators

| Comparison operator | What it means                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>==</code>     | <b>Equal to.</b> Compares two values to see if they're the same. Can be used to compare numbers or strings.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
| <code>!=</code>     | <b>Not equal to.</b> Compares two values to see if they're not the same. Can be used to compare numbers or strings.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| <code>===</code>    | <b>Strict equal to.</b> Compares not only the values but also the type of the value. In other words, the two values must also share the same type—string, number, or Boolean—in order for the condition to be true. For example, while <code>'2'===2</code> is true, <code>'2'==2</code> is not true, because the first value is inside quote marks (a string) and the second is a number. You should be careful with this operator, since values in forms—even numbers—are always strings, so using strict equality to compare a number retrieved from a form field to a number using strict equality ( <code>'2'===2</code> ) will be false. |
| <code>!==</code>    | <b>Strict not equal to.</b> Like strict equal to compare values and type. For example, while <code>'2'!=2</code> is false, <code>'2'!==2</code> is true, because although the values are the same, the types are not.                                                                                                                                                                                                                                                                                                                                                                                                                          |
| <code>&gt;</code>   | <b>Greater than.</b> Compares two numbers and checks if the number on the left side is greater than the number on the right. For example, <code>2 &gt; 1</code> is true, since 2 is a bigger number than 1, but <code>2 &gt; 3</code> is false, since 2 isn't bigger than 3.                                                                                                                                                                                                                                                                                                                                                                   |
| <code>&lt;</code>   | <b>Less than.</b> Compares two numbers and checks if the number on the left side is less than the number on the right. For example, <code>2 &lt; 3</code> is true, since 2 is a smaller number than 3, but <code>2 &lt; 1</code> is false, since 2 isn't less than 1.                                                                                                                                                                                                                                                                                                                                                                          |
| <code>&gt;=</code>  | <b>Greater than or equal to.</b> Compares two numbers and checks if the number on the left side is greater than or the same value as the number on the right. For example, <code>2 &gt;= 2</code> is true, since 2 is the same as 2, but <code>2 &gt;= 3</code> is false, since 2 isn't a bigger number 3, nor is it equal to 3.                                                                                                                                                                                                                                                                                                               |
| <code>&lt;=</code>  | <b>Less than or equal to.</b> Compares two numbers and checks if the number on the left side is less than or the same value as the number on the right. For example, <code>2 &lt;= 2</code> is true, since 2 is the same as 2, but <code>2 &lt;= 1</code> is false, since 2 isn't a smaller number than 1, nor is 2 equal to 1.                                                                                                                                                                                                                                                                                                                |

# If else - Friday night planner

---

```
var fridayCash = prompt('How much money can you spend?', '');

if (fridayCash >= 50) {
 alert('You should go out to a dinner and a movie.');} else if (fridayCash >= 35) {
 alert('You should go out to a fine meal.');} else if (fridayCash >= 12) {
 alert('You should go see a movie.');} else {
 alert('Looks like you will be watching TV.');}
```

# Switch cases

---

```
var fruitType = prompt('enter a fruit', '');
switch (fruitType) {
 case 'Oranges':
 console.log('Oranges are $0.59 a pound. ');
 break;
 case 'Apples':
 console.log('Apples are $0.32 a pound. ');
 break;
 case 'Bananas':
 console.log('Bananas are $0.48 a pound. ');
 break;
 default:
 console.log(`Sorry, we are out of ${fruitType}.`);
}
console.log("Is there anything else you'd like?");
```

# While loops

---

```
var days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'];
var counter = 1;
 while (counter < days.length) {
 document.write(days[counter] + ', ');
 counter++;
 }
```

# For loops

---

```
var days = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday'];
 for (var i=0; i<days.length; i++) {
 document.write(days[i] + ', ');
 }
```

# Functions

---

## Program to print today's date

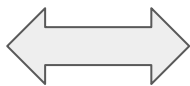
```
function printToday() {
 var today = new Date();
 document.write(today.toString());
}
```



# Arrow Functions

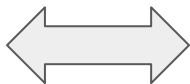
---

```
// Traditional Anonymous
Function
function (a, b){
 return a + b + 100;
}
```



```
// Arrow Function
(a, b) => a + b + 100;
```

```
// Traditional Anonymous
Function (no arguments)
let a = 4;
let b = 2;
function (){
 return a + b + 100;
}
```



```
// Arrow Function (no arguments)
let a = 4;
let b = 2;
() => a + b + 100;
```

# this

---

- The this keyword refers to a special property of an execution context.
- A function's this keyword behaves a little differently in JavaScript compared to other languages.
- It also has some differences between strict mode and non-strict mode.
- **Global Context:**
  - this refers to the global object whether in strict mode or not.

```
// In web browsers, the window object is also the global object:
console.log(this === window); // true

a = 37;
console.log(window.a); // 37

this.b = "MDN";
console.log(window.b) // "MDN"
console.log(b) // "MDN"
```

# this

---

- Function Context:
  - the value of this depends on how the function is called.

## Not in **strict** mode

```
function f1() {
 return this;
}

// In a browser:
f1() === window; // true
```

## In **strict** mode

```
function f2() {
 'use strict'; // see strict mode
 return this;
}

f2() === undefined; // true
```

# Functions and function expressions

---

## function

```
function square(number) {
 return number * number;
}
```

## Function expression

```
const square = function(number) { return number *
 number }
var x = square(4) // x gets the value 16
```

- Function expressions can be anonymous
- a name can be provided with a function expression.

```
const factorial = function fac(n) { return n < 2 ? 1 : n * fac(n - 1) }

console.log(factorial(3))
```

# How to handle errors

---

## try...catch statement

```
function getMonthName(mo) {
 mo = mo - 1; // Adjust month number for array index (1 = Jan, 12 = Dec)
 let months = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul',
 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];
 if (months[mo]) {
 return months[mo];
 } else {
 throw 'InvalidMonthNo'; // throw keyword is used here
 }
}

try { // statements to try
 monthName = getMonthName(myMonth); // function could throw exception
}
catch (e) {
 monthName = 'unknown';
 logMyErrors(e); // pass exception object to error handler (i.e. your own
function)
}
```

# How to handle errors

---

## Utilizing Error Objects

```
function doSomethingErrorProne() {
 if (ourCodeMakesAMistake()) {
 throw (new Error('The message'));
 } else {
 doSomethingToGetAJavascriptError();
 }
}
:
try {
 doSomethingErrorProne();
} catch (e) {
 // NOW, we actually use `console.error()`
 console.error(e.name); // logs 'Error'
 console.error(e.message); // logs 'The message', or a JavaScript error message
}
```



# Document Object Model

A programming interface for web documents

# Document Object Model

- HTML tags define a hierarchical structure called the Document Object Model, or **DOM** for short.

```
<!doctype html>
```

```
<html>
```

```
 <head>
```

```
 <title>Hello World!</title>
```

```
 </head>
```

```
 <body>
```

```
 <h1>Hello World!</h1>
```

```
 <div>
```

```

```

```
 List Item
```

```
 List Item
```

```
 List Item
```

```

```

```
 <p>This is a paragraph.</p>
```

```
 <p>This is a second paragraph.</p>
```

```
 </div>
```

```

```

```
 List Item 1
```

```
 List Item 2
```

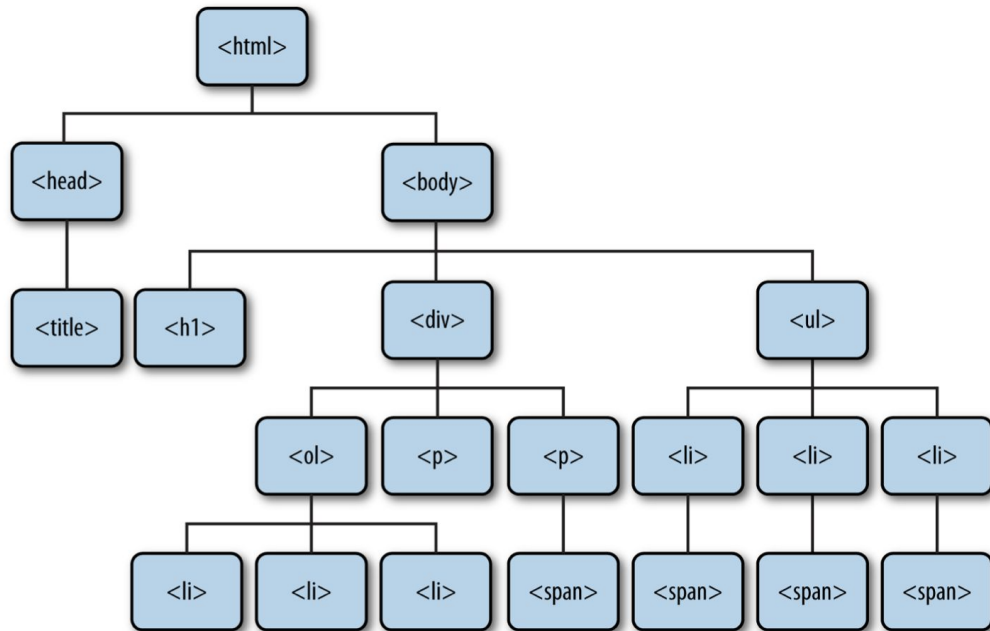
```
 List Item 3
```

```

```

```
 </body>
```

```
</html>
```





# DOM and JavaScript

---

- **What** is DOM?
  - An application programming interface (API) for manipulating HTML documents.
- **What does it represent?**
  - represents an HTML document as a tree of nodes.
- **Why** do we need DOM?
  - provides functions that allow you to add, remove, and modify parts of the document effectively.
  - DOM also provides cross-platform and language-independent ways of manipulating HTML and XML documents



# Selecting Elements

---

- `getElementById()` – select an element by id.
- `getElementsByName()` – select elements by name.
- `getElementsByTagName()` – select elements by a tag name.
- `getElementsByClassName()` – select elements by one or more class names.
- `querySelector()` – select elements by CSS selectors.

# Manipulating Elements

---

- `createElement()` – create a new element.
- `appendChild()` – append a node to a list of child nodes of a specified parent node.
- `textContent` – get and set the text content of a node.
- `innerHTML` – get and set the HTML content of an element.
- `insertBefore()` – insert a new node before an existing node as a child node of a specified parent node.
- `insertAfter()` helper function – insert a new node after an existing node as a child node of a specified parent node.
- `append()` – insert a node after the last child node of a parent node.
- `prepend()` – insert a node before the first child node of a parent node.
- `insertAdjacentHTML()` – parse a text as HTML and insert the resulting nodes into the document at a specified position.
- Ex: DOMJS1



# Events

Actions or occurrences that happen in system

# Events examples

---

- The user selects a certain element or hovers the cursor over a certain element.
- The user chooses a key on the keyboard.
- The user resizes or closes the browser window.
- A web page finishes loading.
- A form is submitted.
- A video is played, paused, or finishes.
- An error occurs.

# Example - JSEvents

---

- The recommended mechanism for adding event handlers in web pages is the `addEventListener()`:
- specify two parameters:
  - the name of the event we want to register this handler for, and
  - the code that comprises the handler function we want to run in response to it.

```
const btn = document.querySelector('button');

function random(number) {
 return Math.floor(Math.random() * (number+1));
}

btn.addEventListener('click', () => {
 const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
 document.body.style.backgroundColor = rndCol;
});
```

# Example - JSEvents (Alternate)

---

It is fine to make the handler function a separate named function

```
const btn = document.querySelector('button');

function random(number) {
 return Math.floor(Math.random() * (number+1));
}

function changeBackground() {
 const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
 document.body.style.backgroundColor = rndCol;
}

btn.addEventListener('click', changeBackground);
```



# Event Handler Properties

---

Objects that can fire events also usually have properties whose name is `on` followed by the name of the event.

```
const btn = document.querySelector('button');

function random(number) {
 return Math.floor(Math.random() * (number+1));
}

function bgChange() {
 const rndCol = `rgb(${random(255)}, ${random(255)}, ${random(255)})`;
 document.body.style.backgroundColor = rndCol;
}

btn.onclick = bgChange;
```

# Other event examples

---

- Mouse Events:

- click
- dblclick
- mousedown
- mouseup
- mouseover
- mouseout

- Document/Window Events:

- load
- resize
- scroll
- unload

- Keyboard Events:

- keypress
- keydown
- keyup

- Form Events:

- submit
- reset
- change
- focus
- blur

# Simplest JavaScript Examples

Examples

# How/where to add JavaScript

---

- Internal JavaScript

- In HTML document — either within head tag or after body tag:

```
<script>
```

```
 // JavaScript goes here
```

```
</script>
```

- External JavaScript

- Create a new file with .js extension and write JavaScript code in new file

In HTML, include following:

```
<script src="script.js" defer></script>
```

# Example 1 - JS1



<p>Player 1: X</p>

```
p {
 font-family: 'helvetica neue', helvetica, sans-serif;
 letter-spacing: 1px;
 text-transform: uppercase;
 text-align: center;
 border: 2px solid rgba(0,0,200,0.6);
 background: rgba(0,0,200,0.3);
 color: rgba(0,0,200,0.6);
 box-shadow: 1px 1px 2px rgba(0,0,200,0.4);
 border-radius: 10px;
 padding: 3px 10px;
 display: inline-block;
 cursor: pointer;
}
```

```
const para = document.querySelector('p');

para.addEventListener('click', updateName);

function updateName() {
 const name = prompt('Enter a new name');
 para.textContent = `Player 1: ${name}`;
}
```

# Example JS2



`<button>Click me</button>`

```
function createParagraph() {
 const para = document.createElement('p');
 para.textContent = 'You clicked the button!';
 document.body.appendChild(para);
}

const buttons = document.querySelectorAll('button');

for (const button of buttons) {
 button.addEventListener('click', createParagraph);
}
```

# Example 3 - NumberGuessingGame

---

- Number guessing game

## Number guessing game

We have selected a random number between 1 and 100. See if you can guess it in 10 turns or fewer. We'll tell you if your guess was too high or too low.

Enter a guess:

Submit guess

# Number guessing steps

---

1. Generate a random number between 1 and 100.
2. Record the turn number the player is on. Start it on 1.
3. Provide the player with a way to guess what the number is.
4. Once a guess has been submitted first record it somewhere so the user can see their previous guesses.
5. Next, check whether it is the correct number.
6. If it is correct:
  - Display congratulations message.
  - Stop the player from being able to enter more guesses (this would mess the game up).
  - Display control allowing the player to restart the game.
7. If it is wrong and the player has turns left:
  - Tell the player they are wrong and whether their guess was too high or too low.
  - Allow them to enter another guess.
  - Increment the turn number by 1.
8. If it is wrong and the player has no turns left:
  - Tell the player it is game over.
  - Stop the player from being able to enter more guesses (this would mess the game up).
  - Display control allowing the player to restart the game.
9. Once the game restarts, make sure the game logic and UI are completely reset, then go back to step 1.



# References and Citations for the some slides and examples

---

- <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>
- [https://www.w3schools.com/cssref/sel\\_firstline.asp](https://www.w3schools.com/cssref/sel_firstline.asp)

## Citation for MDN documents

- About MDN by Mozilla Contributors is licensed under [CC-BY-SA 2.5](https://creativecommons.org/licenses/by-sa/2.5/).
- Any copyright is dedicated to the Public Domain.  
<http://creativecommons.org/publicdomain/zero/1.0/>