CIS 371 Web Application Programming

Midterm Review

Lecturer: Dr. Yong Zhuang

Based on the original version by Professor Hans Dulimarta.

Cascading Style Sheets (CSS)

Applying CSS to HTML

```
<html>                    Option 1: External
  <head>
    <link rel="stylesheet"
          href="mystyles.css">
  </head>
  <body>
    <p>Hello world</p>
  </body>
</html>
```

```
/* in mystyles.css */
p {
  border: 2px
  solid red;
}
```

```
<html>     Option 2: Internal
  <head>
    <style>
      p {
        border: 2px solid red;
      }
    </style>
  </head>
  <body>
    <p>Paragraph 1</p>
    <p>Paragraph 2</p>
  </body>
</html>
```

```
<body>     Option 3: Inline (Not recommended)
  <!-- inline style -->
  <p style="border: 2px solid red">......</p>
</body>
```

CSS “selectors” / “filters”

Various options to select portions(s) of the DOM tree. Select by:

- ID, tag name, CSS class (or combination of them)
- Attribute (with or without its value)
- Parent/Child relationship in the DOM tree, such as
  - All immediate children of ____
  - Any descendants of _____
  - The last grandchild of _____
  - and so on…
- Sibling relationship in the DOM tree
- Permutations of all the above selectors

Selector Permutations: tag & class

```
/* in CSS */
li.fruit {
    color: red
}
```

Apply only to list items with class .fruit

1. Strawberry
2. Raspberry Pi
3. Halle Berry

```
<!-- in HTML -->
<ol>
    <li class="fruit">Strawberry</li>
    <li class="device">RaspBerry Pi</li>
    <li>Halle Berry</li>
</ol>
```

```
/* in CSS */
li .fruit {
    color: red
}
```

Beware of SPACE.

This rule applies to **descendants** of <li> NOT <li> themselves.

1. Strawberry
2. Raspberry Pi
3. Halle Berry

Selector Permutations: tag & attribute

```
/* in CSS */
li[class] {
      color: red
}
```

Apply only to list items with **class** attribute set, regardless of its value

```
<!-- in HTML -->
<ol>
    <li class="fruit">Strawberry</li>
    <li class="device">RaspBerry Pi</li>
    <li>Halle Berry</li>
</ol>
```

1. Strawberry
2. Raspberry Pi
3. Halle Berry

Selector Permutations: tag & attr & attr-value

```
/* in CSS */
li[class*=t] {
      color: red
}
```

Apply only to list items with **class** attribute value containing "**t**"

```
<!-- in HTML -->
<ol>
    <li class="fruit">Strawberry</li>
    <li class="device">RaspBerry Pi</li>
    <li class="actor">Halle Berry</li>
</ol>
```

1. Strawberry
2. Raspberry Pi
3. Halle Berry

Selectors: by attribute(s)

Objective: select elements with a particular attribute

Selectors

- [attr] ⇒ select elements that have attribute attr (regardless of its value)
- [attr=val] ⇒ select elements whose attribute attr is set to “val”
- [attr~=val] ⇒ select elements whose attribute attr **contains** “val” (whole word)
- [attr*=val] ⇒ select elements whose attribute attr **contains** “val” (partial word)
- [attr|=val] ⇒ select elements whose attribute attr **starts with** “val” (whole word)
- [attr^=val] ⇒ select elements whose attribute attr **starts with** “val” (partial word)
- [attr$=val] ⇒ select elements whose attribute attr **ends with** “val” (partial word)
- [attr1=val2][attr2*=val2] ⇒ use multiple attributes (**logical and**)

Selector by relative placement in DOM tree

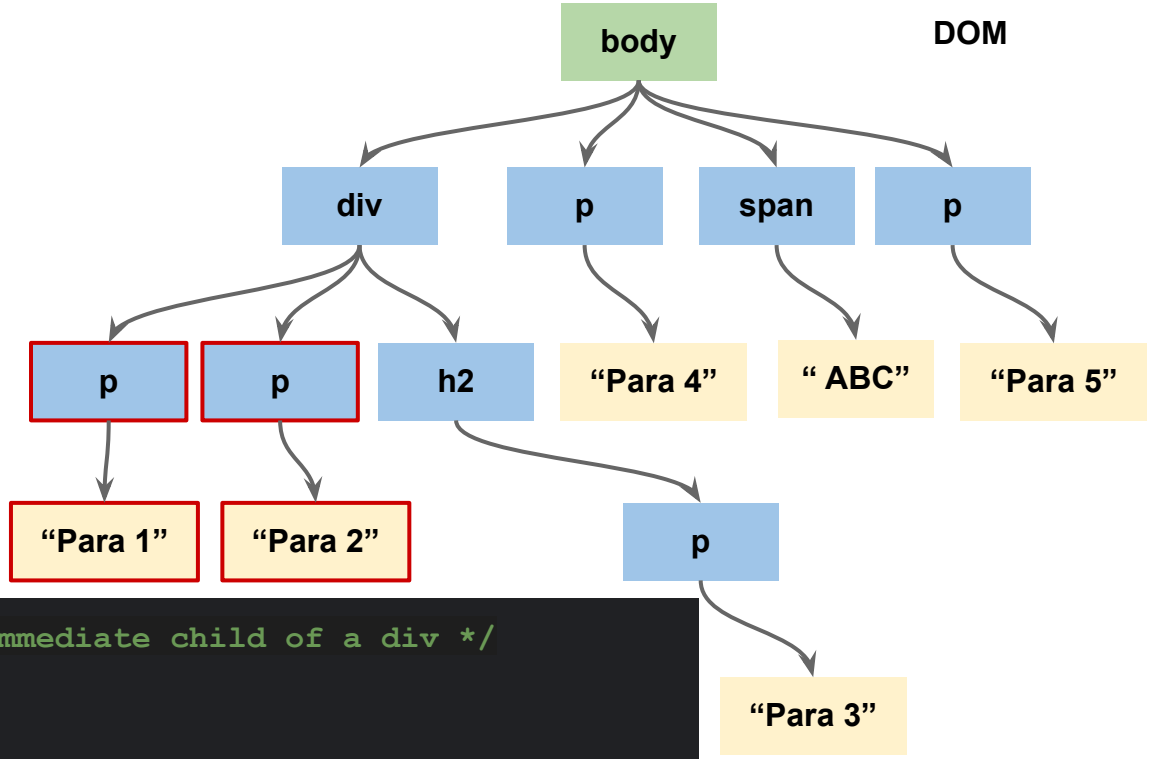
Descendant/Younger Sibling Selectors

Types	Selector	Apply Rules to
Immediate children	`div > p { ...rules...}`	paragraphs which are an immediate children of div
Any descendant	`div  p { ...rules...}`	paragraphs which are a descendant of a div (immediate children included)
Immediate (**younger**) sibling	`div + p { ...rules...}`	one paragraph (immediate younger sibling of a div)
Any **younger** sibling	`div ~ p { ...rules...}`	paragraphs which are younger sibling of a div (immediate sibling included)

Examples

Child (Immediate Descendant) Selector

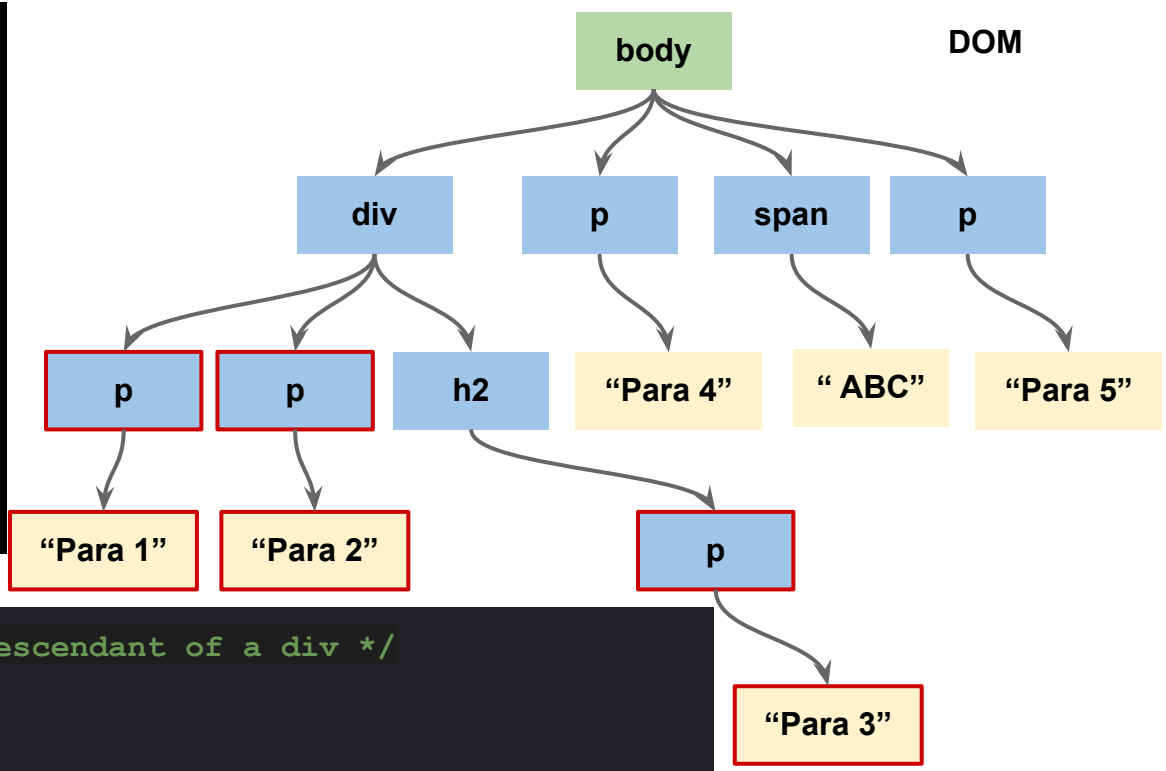
```
<body>
    <div>
        <p>Para 1</p>
        <p>Para 2</p>
        <h2><p>Para 3</p></h2>
    </div>
    <p>Para 4</p>
    <span>ABC</span>
    <p>Para 5</p>
</body>
```




```
/* apply to paragraphs which are an immediate child of a div */
div > p {
    border: 2px solid red;
}
```

(Deeper) Descendant Selector

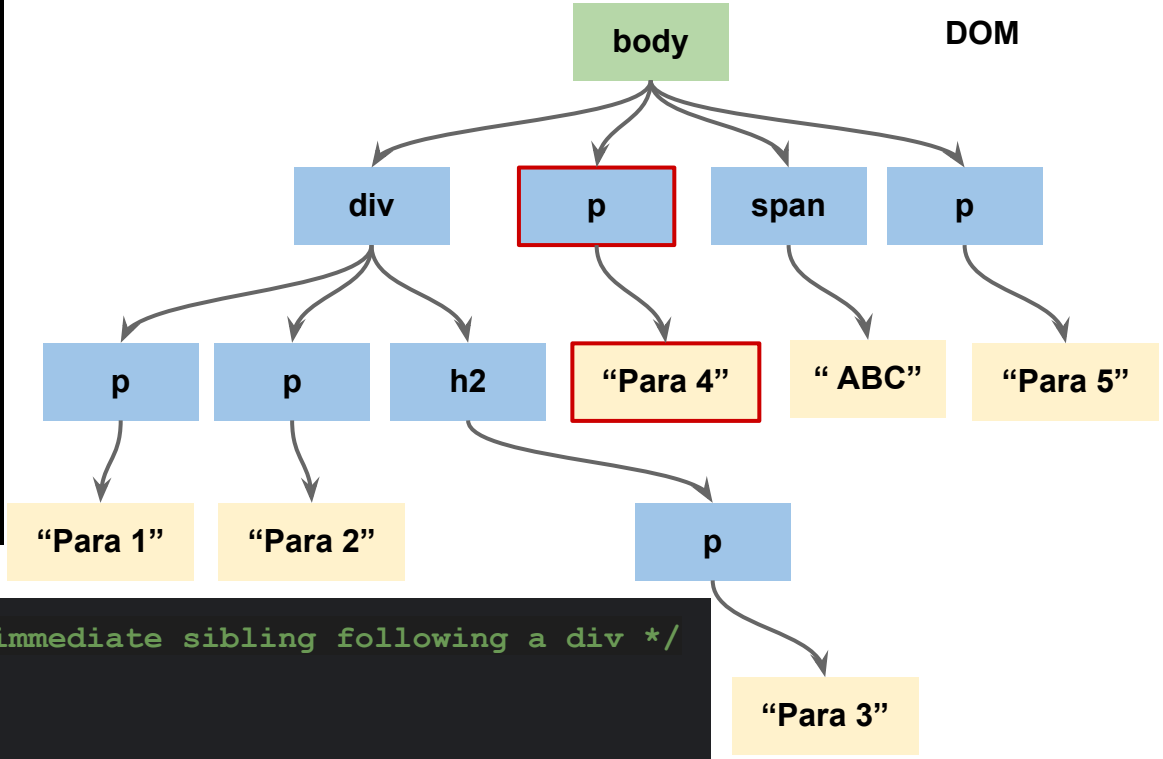
```
<body>
    <div>
        <p>Para 1</p>
        <p>Para 2</p>
        <h2><p>Para 3</p></h2>
    </div>
    <p>Para 4</p>
    <span>ABC</span>
    <p>Para 5</p>
</body>
```




```
/* apply to paragraphs which are a descendant of a div */
div  p {
    border: 2px solid red;
}
```

Immediate Sibling Selector

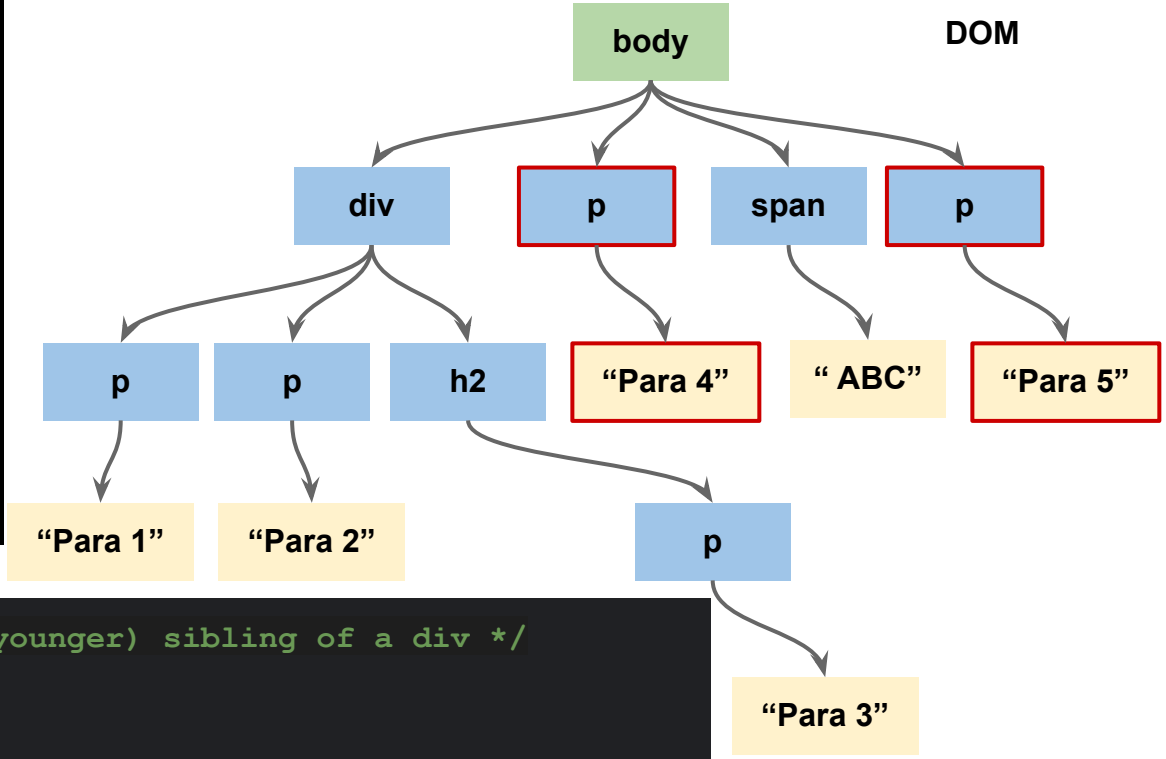
```
<body>
    <div>
        <p>Para 1</p>
        <p>Para 2</p>
        <h2><p>Para 3</p></h2>
    </div>
    <p>Para 4</p>
    <span>ABC</span>
    <p>Para 5</p>
</body>
```




```
/* apply to paragraphs which are an immediate sibling following a div */
div + p {
    border: 2px solid red;
}
```

General Siblings Selector

```
<body>
    <div>
        <p>Para 1</p>
        <p>Para 2</p>
        <h2><p>Para 3</p></h2>
    </div>
    <p>Para 4</p>
    <span>ABC</span>
    <p>Para 5</p>
</body>
```




```
/* apply to paragraphs which are a (younger) sibling of a div */
div ~ p {
    border: 2px solid red;
}
```

Selector Modifiers :pseudo-classes

- Links (:link, :visited, :hover, :active)
- Input (:checked, :disabled, :enabled, :focus, :in-range, :out-of-range, :invalid, :valid, :optional, :required, :read-only, :read-write)
- Child order (:first-child, :last-child, :nth-child, :nth-last-child, :only-child)
- Of-Type order (:first-of-type, :last-of-type, :nth-of-type, :nth-last-of-type, :only-of-type)
- Online reference (look for “Pseudo-classes” on the left)

:first-child vs. :first-of-type

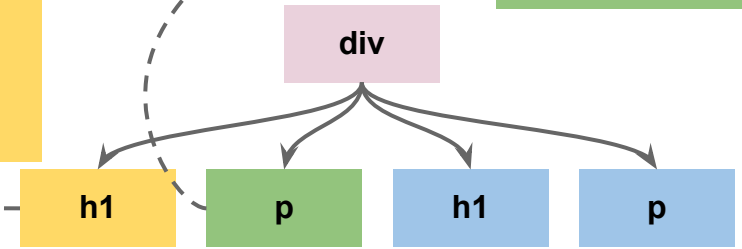
```
<div>
    <h1>First Heading</h1>
    <p>One paragraph</p>
    <h1>Second Heading</h1>
    <p>A bit longer paragraph</p>
</div>
```

```
div p:first-child {
    /* no matching element */
      color: red
}
```

The **first** “daugher” in a family may be the **third** “child”

```
div p:first-type {
    /* no matching element */
      color: red
}
```

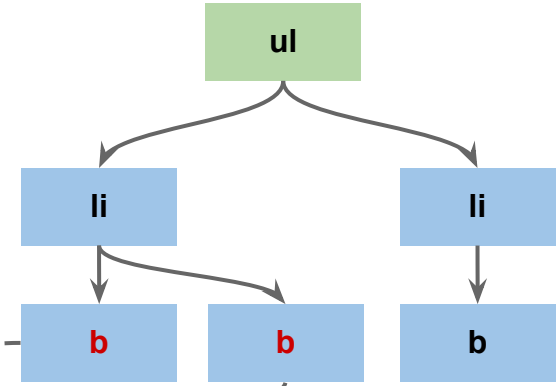
```
div h1:first-child {
    /* no matching element */
      color: red
}
```



:first-child

```
<ul>
    <li>Test <b>one</b> and <b>two</b></li>
    <li>Another <b>text</b></li>
</ul>
```

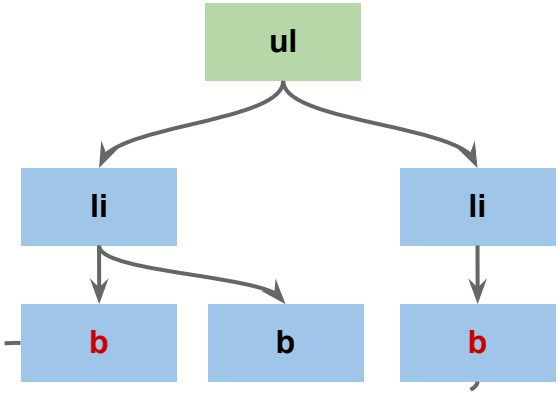
```
li:first-child b{
      color: red;
}
```



:first-child

```
<ul>
    <li>Test <b>one</b> and <b>two</b></li>
    <li>Another <b>text</b></li>
</ul>
```

```
li b:first-child{
      color: red;
}
```



CSS3 :nth-child()

- :nth-child(4): select child #4
- :nth-child(odd): select children #1, #3, #5, …
- :nth-child(even): select children #2, #4, #6, …
- :nth-child(3n+1): select children #1, #4, #7, #10, ...

CSS Grid & Flexbox

A Complete Guide to Grid

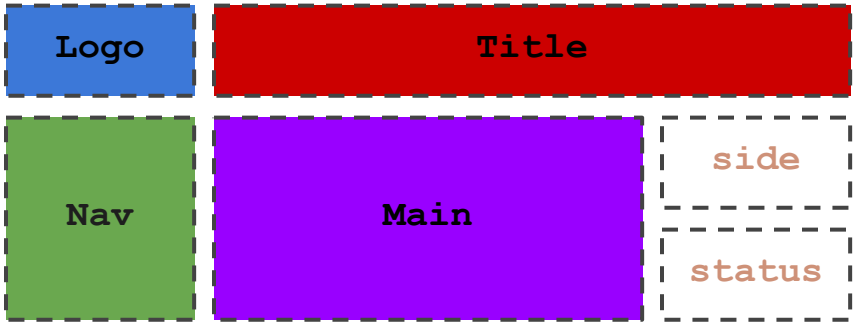
A Complete Guide to Flexbox

Explicit positioning by Area Names

```html
/* in HTML  */
<div id="mybox">
    <span class="blue">Logo</span>
    <span class="red">Title</span>
    <span class="green">Nav</span>
    <span class="purple">Main</span>
</div>
```

```css
#mybox {
    display: grid;
    grid-template-rows: 1fr 1fr 1fr;
    grid-template-columns: 1fr 2fr 1fr;
    grid-template-areas:
    "logo title title"
    "nav main side"
    "nav main status";
}
```

```css
.blue {
    background: blue;
    grid-area: logo
}

.red {
    background: red;
    grid-area: title
}
.green {
    background: green;
    grid-area: nav;
}
.purple {
    background: purple;
    grid-area: main;
}
```



Data Types

Java	TypeScript
boolean	boolean
char	string
String	string
float, double	number
short, int, long	number
	any (no type checking)
	unknown (strict type checking)

TypeScript: The TypeScript Handbook

TypeScript: Online PlayGround: https://typescriptlang.org/play

TS Unions: multiple types

```
// TypeScript
let a: number | boolean;                    // init to undefined
let b: string | number | null = 6.5;        // current type is number

a = "can't do this";              // Error, can't take a string type
a = false;                        // current type is boolean

console.log(typeof b);            // output "number"

b = "hello";
console.log(typeof b);            // output "string"
```

Use this feature in conjunction with typeof test at runtime

== vs ===

==		===	
5 == “5”	true	5 === “5”	false
0.123 == “0.123”	true	0.123 === “0.123”	false
1 == true	true	1 === true	false
5 == true	false	5 === true	false
0 == false	true	0 === false	false
“0” == false	true	“0” === false	false
“1” == true	true	“1” === true	false
With internal type conversion		***No type conversion***	

Arrays: for, for-in vs. for-of

```
const fruits = ["Apple", "Banana", "Cherry"];
```

```
for (let k = 0; k < fruits.length; k++) {
  console.debug("At", k, fruits[k]);
}
```

```
for (let k in fruits) {
  console.debug("At", k, fruits[k]);
}
```

for-in

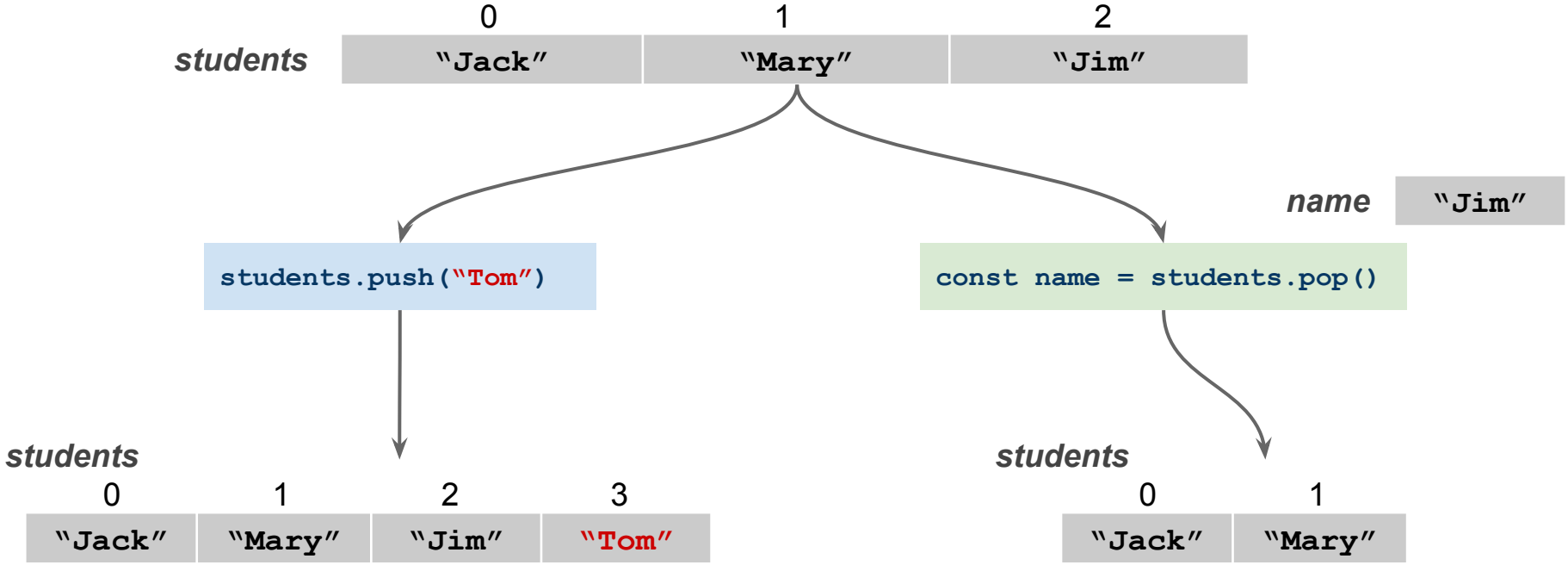
```
for (let f of fruits) {
  console.debug(f);
}
```

for-of

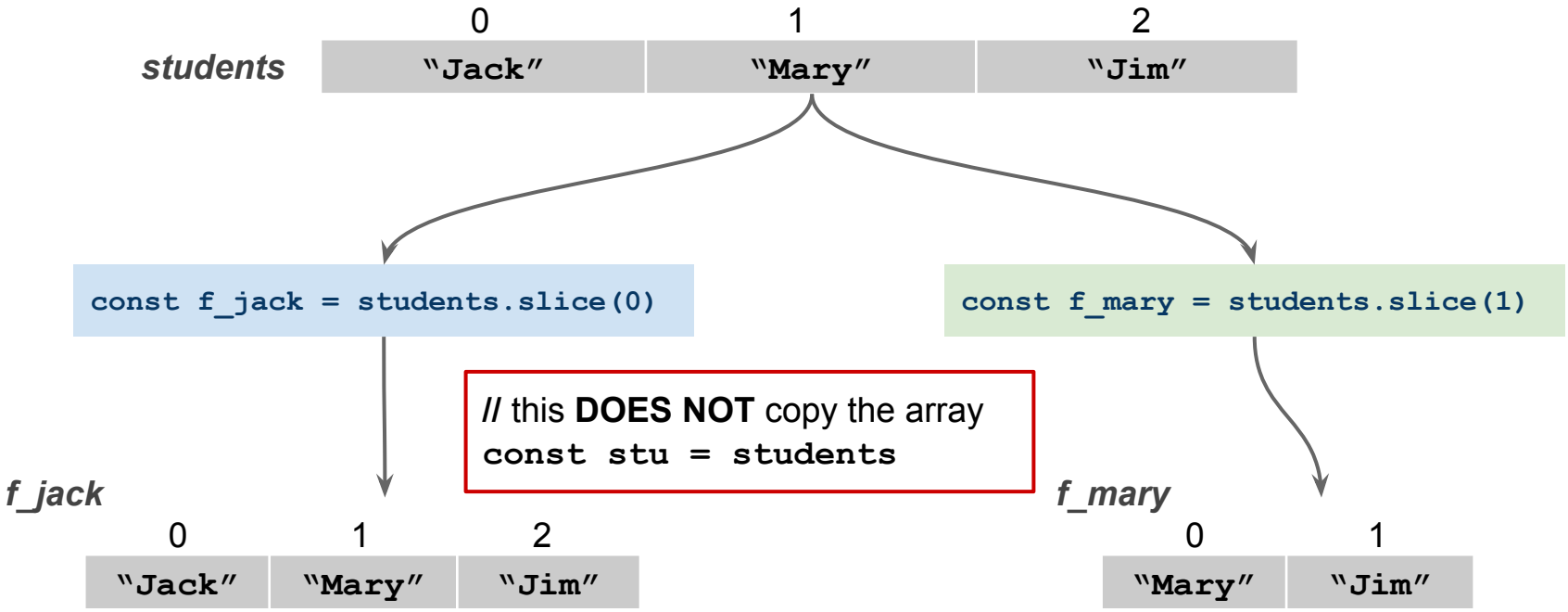
```
At 0 Apple
At 1 Banana
At 2 Cherry
```

```
Apple
Banana
Cherry
```

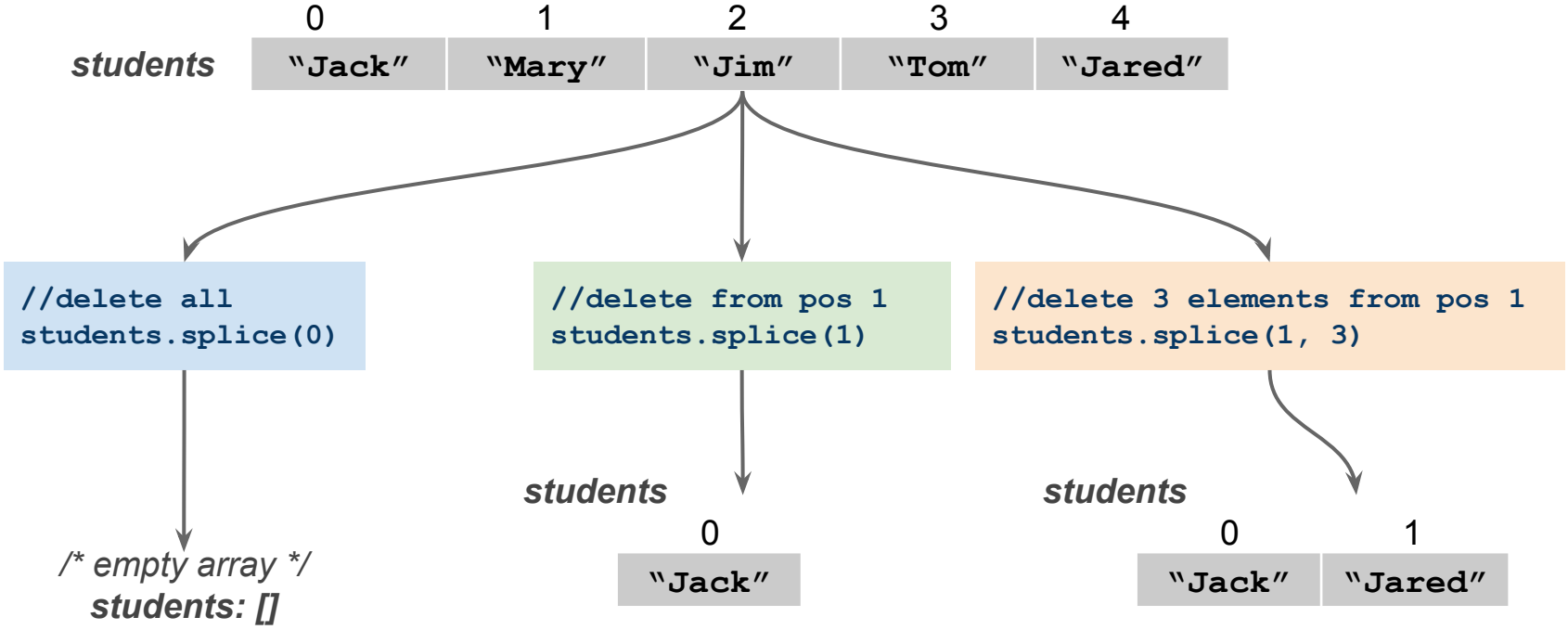
Array: .push() and .pop()



Array: .slice() creates a copy



Array: .splice() delete elements



Array: .splice() replaces elements

students

0	1	2	3	4
"Jack"	"Mary"	"Jim"	"Tom"	"Jared"

```
//delete 3 and insert "Bob"
students.splice(1, 3, "Bob")
```

students

0	1	2
"Jack"	"Bob"	"Jared"

```
//delete 3 elements and insert "Bob" & "Cook"
students.splice(1, 3, "Bob", "Cook")
```

students

0	1	2	3
"Jack"	"Bob"	"Cook"	"Jared"

Objects in TypeScript

```
// Typeless objects
const in_a_month = {
  name: "September",
  days: 30
}

const employee_vacation = {
  name: "Bob",  days: 11
}
```

```
// Typed objects
type Monthly = {
  name: string,
  days: number
}
const in_a_month:Monthly {
  name: "September",
  days: 30
}
```

```
type VacationDays = {
  name: string,
  days: number
}
const employee_vacation:VacationDays = {
  name: "Bob",
  days: 11
}
```

Array of Objects

```java
// In Atom.java
class Atom {
  public String name;
  public weight double;
}

// In AnotherFile.java
ArrayList<Atom> atoms = new ArrayList<>();
Atom a = new Atom("Carbon", 12);
atoms.add(a);
Atom b = new Atom("Oxygen", 16);
atoms.add(b);
atoms.add(new Atom("Natrium", 23);
```

TS: option 1

```typescript
// TypeScript (no class required)
 const atoms = [];
 atoms.push({ name: "Carbon", weight: 12});
 atoms.push({ name: "Oxygen", weight: 16});
 atoms.push({ name: "Natrium", weight: 23});
```

TS: option 2

```typescript
// Or initialize the array
const atoms = [
  { name: "Carbon", weight: 12},
  { name: "Oxygen", weight: 16},
  { name: "Natrium", weight: 23}
];
```

Array of Typed Objects

Typeless array

```
const atoms = [];
atoms.push({ name: "Carbon", weight: 12});
atoms.push({ namme: "Fluor", weight: 12}); // OK
atoms.push({ name: "Oxygen"}); // OK
atoms.push({ name: "Natrium", weight: 23, isMetal: false}); // OK
```

```
// Declare a type
type Atom = {
  name: string,
  weight: number
}
```

Typed array

```
const atoms:Array<Atom> = [];
 atoms.push({ name: "Carbon", weight: 12});
 atoms.push({ namme: "Fluor", weight: 12});        // ERROR: "namme" does not exist
 atoms.push({ name: "Oxygen"});                    // ERROR: property "weight" is missing
 atoms.push({
   name: "Natrium",
   weight: 23,
   isMetal: false});                               // ERROR: "isMetal" does not exist
```

Spreading an Array

```
const primes = [13, 17, 29];
const squares = [9, 25, 81, 144];
```

```
squares.push(primes);
```

```
squares is [9, 25, 81, 144, [13, 17, 19]];
squares.length is 5
```

```
squares.push(...primes);
```

```
// Without spread
for (let p of primes)
  squares.push(p);
```

```
squares is [9, 25, 81, 144, 13, 17, 19];
squares.length is 7
```

Spreading an Object

```
const name = { first: "Bob", last: "Dylan"};
const job = { position: "Web Developer", salary: 75000};
```

```
const one = {name, job};
```

```
{
  name: {
    first: "Bob",
    last: "Dylan"
  },
  job: {
    position: "Web Developer",
    salary: 75000
  }
}
```

```
const two = {name, ... job}
```

```
{
  name: {
    first: "Bob",
    last: "Dylan"
  },
  position: "Web Developer",
  salary: 75000
}
```

```
const three = {
  ... name,
  ... job
}
```

```
{
  first: "Bob",
  last: "Dylan",
  position: "Web Developer",
  salary: 75000
}
```

Spread on Objects (with duplicate props)

```
const prop1 = {name: "Carbon", abbrev: "Cb"}
const prop2 = {weight: 12, abbrev: "C"}
// without spread on prop1
const element = {prop1, ... prop2};
```

Without spread

```
{
  prop1: {
    name: "Carbon", abbrev: "Cb"
  },
  weight: 12, abbrev: "C"
}
```

```
const prop1 = {name: "Carbon", abbrev: "Ca"}
const prop2 = {weight: 12, abbrev: "C", name: "Clue"}
// with spread
const element = {...prop1, ...prop2, isMetal: false};
const el2     = {...prop2, ...prop1, isMetal: false};
```

With spread

```
{
  isMetal: false,
  name: "Clue",
  abbrev: "C",
  weight: 12,
}
```

```
{
  isMetal: false,
  name: "Carbon",
  abbrev: "Ca",
  weight: 12,
}
```

Later values overwrite previous values of the same key

Object spread: copy and modify

```
const bob = {
  first: "Bob",
  last: "Dylan",
  position: "Web Developer",
  salary: 75000
}
```

```
const bob_now = {
  ...bob,
  workFromHome: true,
  position: "Cloud Data Egr.",
  salary: 78000
}
```

bob_now

```
{
  first: "Bob",
  last: "Dylan",
  workFromHome: true,
  position: "Cloud Data Egr.",
  salary: 78000
}
```

This won't work (no copy created).

```
const bob_now = bob;
bob_now.position = "Cloud Data Egr.";
bob_now.salary = 78000;
```

HTML Document CRUD methods/functions

Create	`document.createElement()`, `document.createTextNode()`
Read	`___.getElementById()` // SINGULAR `___.getElementsByTagName()`, // PLURAL `___.getElementsByClassName()` // PLURAL `___.querySelector()` // SINGULAR: search by CSS selectors `___.querySelectorAll()` // PLURAL: search by CSS selectors
Update	`___.appendChild()`
Delete	`___.removeChild()`

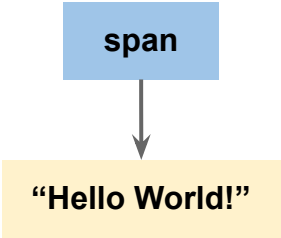
and many more …

```
for (const z in document) {
    if (typeof document[z] === "function") {
        console.log(z);
    }
}
```

Try this yourself

Create Text Nodes

```
<span>Hello world!</span>
```

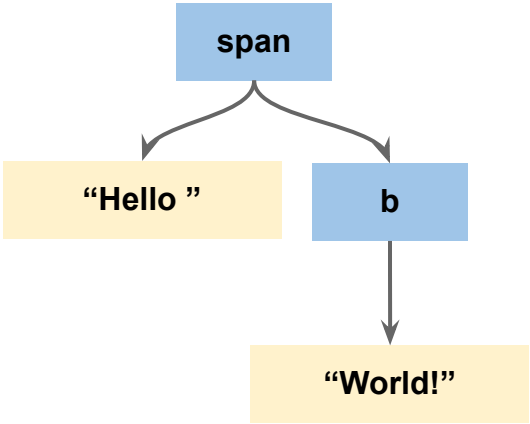



```
// Option 1
const spanParent = document.createElement("span");
const hello = document.createTextNode("Hello World");
spanParent.appendChild(hello);
```

```
// Option 2
const spanParent = document.createElement("span");
spanParent.innerText = "Hello World";
```

Add Multiple Children

```
<span>Hello <b>world!</b></span>
```




```
const spanTop = document.createElement("span");
const txt1 = document.createTextNode("Hello");
spanTop.appendChild(txt1);

const bChild = document.createElement("b");
bChild.innerText = "World";
spanTop.appendChild(bChild);
```

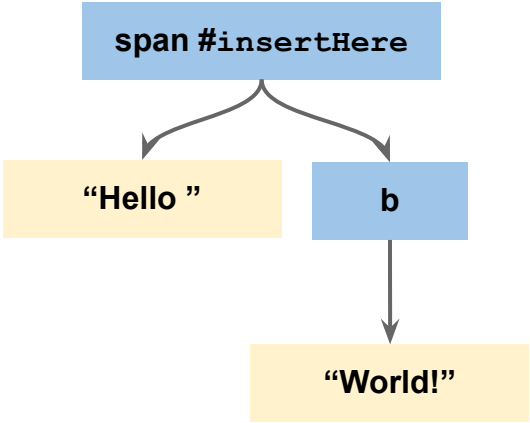
Insert Contents into Existing DOM

before

```
<span id="insertHere"> </span>
```

after

```
<span id="insertHere">
    Hello <b>world!<b>
</span>
```

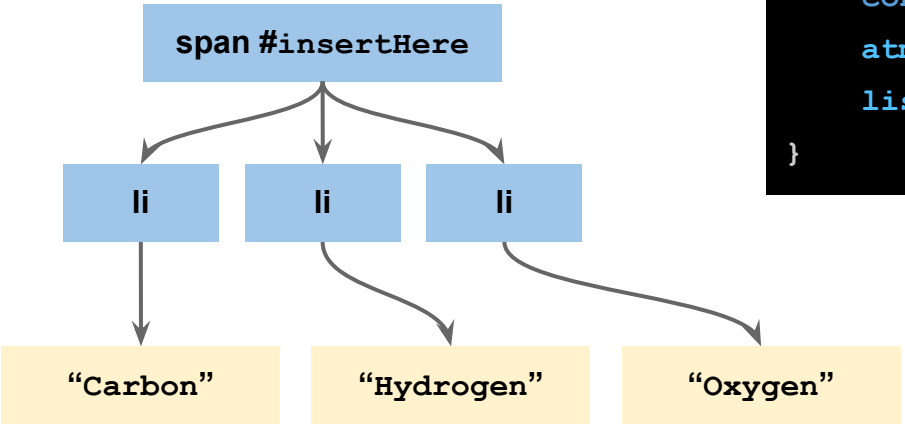



```
const spanTop = document.getElementById("insertHere");
const txt1 = document.createTextNode("Hello");
spanTop.appendChild(txt1);

const bChild = document.createElement("b");
bChild.innerText = "World";
spanTop.appendCHild(bChild);
```

Add Multiple Children from Array

```
<ul>
    <li>Carbon</li>
    <li>Hydrogen</li>
    <li>Oxygen</li>
</ul>
```

```
const atoms = ["Carbon", "Hydrogen", "Oxygen"]
const listTop = document.createElement("ul");

for (let a of atoms) {
    const atm = document.createElement("li");
    atm.innerText = a;
    listTop.appendChild(atm);
}
```



Setting attributes

```
<a id="intro" class="deepIndent noAds" href="http://go.org">
    Some text here
</a>
```

```
const sample = document.createElement("a");
sample.innerText = "Some text here";
sample.id = "intro"
sample.classList.add("deepIndent");
sample.classList.add("noAds");
sample.setAttribute("href", "http://go.org");
```

```
const sample = document.createElement("a");
sample.innerText = "Some text here";
sample.setAttribute("id", "intro");
sample.setAttribute("class", "deepIndent noAds");
sample.setAttribute("href", "http://go.org");
```

querySelector(): select ONE element

```
<body>
    <p class="title">Ice Cream Flavor:</p>
    <ul>
        <li>Death by Chocolate</li>
        <li>Mint Chocolate Chip</li>
        <li>Strawberry</li>
    </ul>
    <script src="ice.ts">
</body>
```

Ice Cream Flavors:

- Too much Chocolate
- Mint Chocolate Chip
- Strawberry

```
const item:Element = document.querySelector("ul > li");
#the first one will be returned
item.textContent = "Too much Chocolate";
```

querySelectorAll(): select MULTIPLE elements

```
<body>
    <p class="title">Ice Cream Flavor:</p>
    <ul>
        <li>Death by Chocolate</li>
        <li>Mint Chocolate Chip</li>
        <li>Strawberry</li>
    </ul>
    <script src="ice.ts">
</body>
```

```
let items:NodeListOf<Element>;
items = document.querySelectorAll("ul > li");
for (let flav of items) {
    if (flav.textContent.includes("Chocolate")){
        flav.textContent = flav.textContent + " (on sale)";
    }
}
```

Ice Cream Flavors:

- Death by Chocolate (on sale)
- Mint Chocolate Chip (on sale)
- Strawberry

CSS Selector and querySelector(All)

```
const q1 = document.querySelector("h2 + p");
q1.classList.add("red");   // Affect "First paragraph"
const q2 = document.querySelector("h2 ~ ol > li:first-child");
q1.classList.add("red");   // Affect "Strawberry"
const q3 = document.querySelector("li:last-child");
q1.classList.add("red");   // Affect "Barry Manilow"
```

```
<body>
  <h2>Some heading</h2>
  <p>First paragraph</p>
  <ol>
    <li class="fruit">Strawberry</li>
    <li class="device">Raspberry Pi</li>
    <li class="singer">Barry Manilow</li>
  </ol>
  <p>Second paragraph</p>
</body>
```

```
const pars = document.querySelectorAll("h2 ~ p");
for (let x of pars) {
    // Apply to "First paragraph" and "Second paragraph"
    x.setAttribute("___", "___");
}
const who = document.querySelectorAll("ol > li.singer");
for (let x of who) {
   // Apply to "Barry Manilow"
}
```

JavaScript Events

Source of Event	Events
Window	onload, onresize, onunload, ...
Document	onkeydown, onkeyup, onmousedown, onmouseup, onmouseenter, onmouseleave, ...
Input field	onblur, onfocus, onchange, ….
Button	onclick, ondblclick
Complete Reference: Event APIs	

Setting Up Event Handlers

- Which Event?
- Who is the event source?
  - Resize => window
  - Key presses => document
  - Load => document
  - Click => button, image, ….
  - Focus => input elements
  - Mouse => elements
- Details of the event object properties (MouseEvent, KeyboardEvent, ….). Refer to online API

```
function keyHandler(ev: KeyboardEvent): void {
    // put code here
}
function clickHandler(ev:MouseEvent): void {
    // put code here
}
document.addEventListener("keypress", keyHandler);
const myLogo = document.getElementById("myLogo");
myLogo.addEventListener("click", clickHandler);
```

addEventListener

inline event attributes

```
<button onclick="clickFunction()">Click Me</button>
```

```
function clickFunction() {
    // put code here
}
```

CodePen: Event Handling Demo

Counting Click

Fat Arrow fns: single-line return contraction

```
const plusTwo = (a:number, b:number) : number => {
    const sum = a + b;
    return sum;
}
```

no 'function' keyword.

```
const plusTwo = (a:number, b:number) : number => {
    return a + b;
}
```

If 'return' can be the only statement

omit both the curly braces { } and the 'return' keyword.

```
const plusTwo = (a:number, b:number) : number => a + b;
const plusTwo = (a,b) => a + b;    // typeless
```

implicit return

Array.sort()

```
const atoms = ["Neon", "Iron", "Calcium", "Hydrogen"]
console.log(atoms.sorted())
// ["Calcium", "Hydrogen", "Iron", "Neon"]

const primes = [23, 17, 5, 101, 19]
const sorted_nums = primes.sort()
console.log(sorted_nums)      // [101, 17, 19, 23, 5]
```

Array.prototype.sort()

The `sort()` method of `Array` instances sorts the elements of an array *in place* and returns the reference to the same array, now sorted. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values.

Online Doc

Array.sort() builtin behavior

```
[23, 17, 5, 101, 19]
```

implicit conversion to string

```
["23", "17", "5", "101", "19"]
```

sort()

```
["101", "17", "19", "23", "5"]
```

To fix this ***"bug"***, we have to tell sort() the **collating order between two data items**

Array.sort() with collating order

```
function numericOrder(a:number, b:number): number {
    if (a < b) return -1;              // any negative number
    else if (a > b) return +1;         // any positive number
    else return 0;
}

const primes = [23, 17, 5, 101, 19]
const sorted_nums = primes.sort(numericOrder)
console.log(sorted_nums) // [5, 17, 19, 23, 101] Ok
```

*The collating function must return a **number***

- *Negative when the "first" item should be placed BEFORE the "second" item*
- *Positive when the "first" item should be placed AFTER the "second" item*
- *Zero when the order of the two items is irrelevant*

Array.sort() on objects

```
type Language = {
    name: string;  yearCreated: number
}
const langs: Language[] = [
    { name: "C", yearCreated: 1970},
    { name: "JavaScript", yearCreated: 1995},
    { name: "Fortran", yearCreated: 1954}
]
function orderByName(a:Language, b:Language): number {
    return a.name.localeCompare(b.name)
}
function orderByYear(a:Language, b:Language): number {
    return a.yearCreated - b.yearCreated
}
langs.sort(orderByName)
```

*The collating function takes two parameters of type Language but must **return a number***

Array high-order functions

- Array.every(), Array.some()
- Array.find(), findIndex()
- **Array.filter(), Array.map(), Array.flatmap()**
- Array.forEach()
- **Array.reduce()**
- … and many others