Document Object Model (DOM)

Browser JavaScript interface to HTML document

- HTML document exposed as a collection of JavaScript objects and methods

  The **D**ocument **O**bject **M**odel (DOM)

- JavaScript can query or modify the HTML document
- Accessible via the JavaScript global scope, aliases:

  `window`

  `this` (When not using `'use strict';`)

DOM hierarchy

- Rooted at `window.document (html tag)`
- Follows HTML document structure

  `window.document.head`

  `window.document.body`

- Tree nodes (DOM objects) have tons (~250) of properties, most private

Objects (representing elements, raw text, etc.) have a common set of properties and methods called a DOM "Node"

DOM Node properties and methods

- Identification

  `nodeName` property is element type (uppercase: `P`, `DIV`, etc.) or `#text`

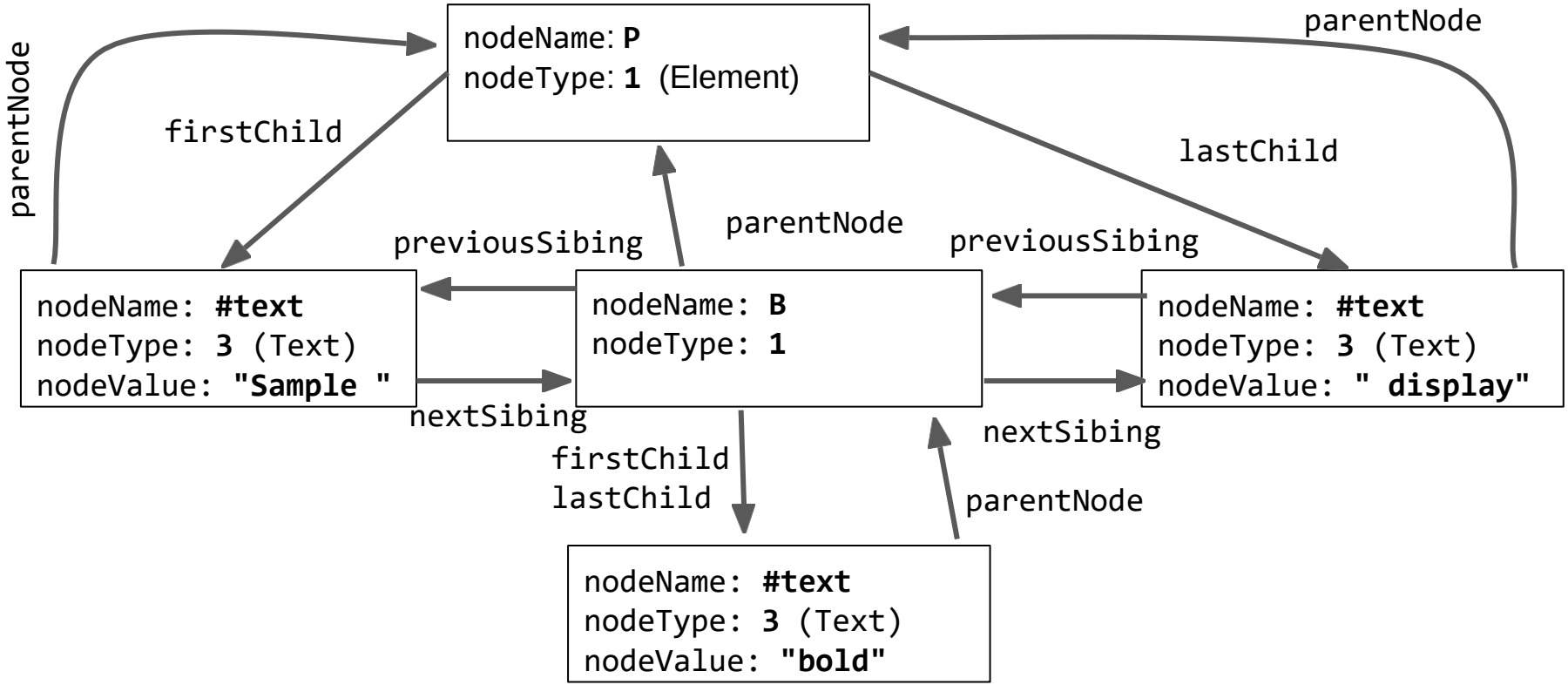
- Encode document's hierarchical structure

  `parentNode`, `nextSibling`, `previousSibling`, `firstChild`, `lastChild`.

- Provide accessor and mutator methods

  E.g. `getAttribute`, `setAttribute` methods, etc..

Sample bold display



Accessing DOM Nodes

- Walk DOM hierarchy (not recommended)

```
element = document.body.firstChild.nextSibling.firstChild;
element.setAttribute(…
```

- Use DOM lookup method. An example using ids:

HTML: `<div id="div42">...</div>`

```
element = document.getElementById("div42");
element.setAttribute(…
```

- Many: `getElementsByClassName()`, `getElementsByTagName()`, …
  - Can start lookup at any element:

```
document.body.firstChild.getElementsByTagName()
```

More commonly used Node properties/methods

- `textContent` - text content of a node and its descendants

  Previous slide example: P Node `textContext` is "Sample bold display"

- `innerHTML` - HTML syntax describing the element's descendants.

  Previous slide example: P Node `innerHTML` is "Sample bold display"

- `outerHTML` - similar but includes element " Sample bold display "

- `getAttribute()/setAttribute()` - Get or set the attribute of an element

Common DOM mutating operations

- Change the content of an element

  ```
  element.innerHTML = "This text is <i>important</i>";
  ```

  Replaces content but retains attributes. DOM Node structure updated.

- Change an `<img` tag `src` attribute (e.g. toggle appearance on click)

  ```
  img.src="newImage.jpg";
  ```

- Make element visible or invisible (e.g., for expandable sections, modals)

  Invisible: `element.style.display = "none";`
  Visible: `element.style.display = "";`

DOM and CSS interactions

- Can update an element's class

  ```
  element.className = "active";
  ```

- Can update element's style

  ```
  element.style.color = "#ff0000";     // Not preferred way!
  ```

- Can also query DOM by CSS selector

  `document.querySelector()` and `document.querySelectorAll()`

Changing the Node structure

- Create a new element (can also `cloneNode()` an existing one)

```
element = document.createElement("P");
            or
element = document.createTextNode("My Text");
```

- Add it to an existing one

```
parent.appendChild(element);
     or
parent.insertBefore(element, sibling);
```

- Can also remove Nodes: `node.removeChild(oldNode);`

- But, setting innerHTML can be simpler and more efficient.

More DOM operations

- Redirect to a new page

  ```
  window.location.href = "newPage.html";
  ```

  Note: Can result in JavaScript script execution termination

- Communicating with the user

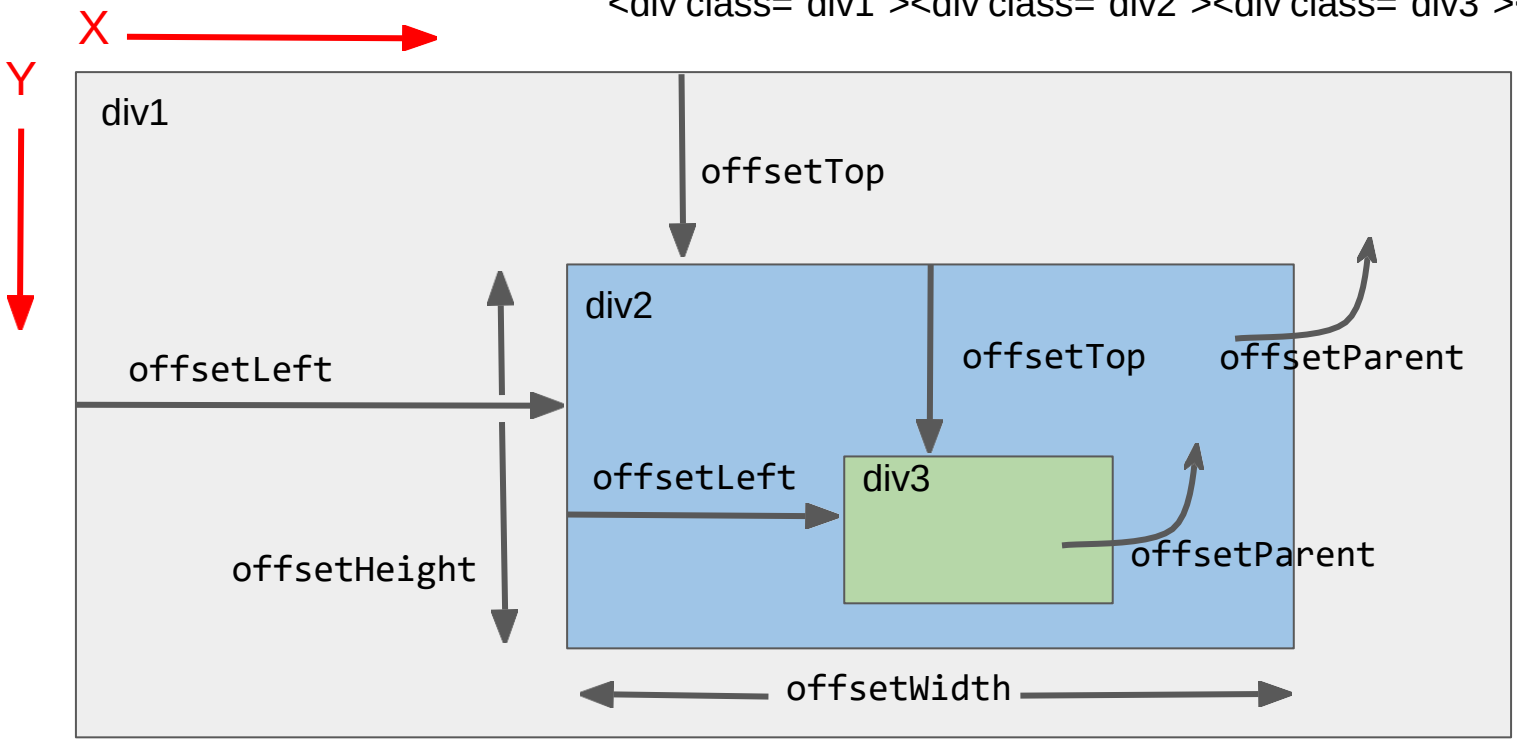
  ```
  console.log("Reached point A");    // Message to browser log

  alert("Wow!");  confirm("OK?");    // Popup dialog
  ```

DOM's Coordinate System

- The screen origin is at the upper left; y increases as you go down
- The position of an element is determined by the upper-left outside corner of its margin
- Read location with `element.offsetLeft`, `element.offsetTop`
- Coordinates are relative to `element.offsetParent`, which is not necessarily the same as `element.parentNode`

DOM Coordinates

/



Positioning elements

- Normally elements are positioned automatically by the browser as part of the document

- To pull an element out of the document flow and position it explicitly:

```
element.style.position = "absolute";  // anything but "static"
element.style.left = "40px";
element.style.top = "10px";
```

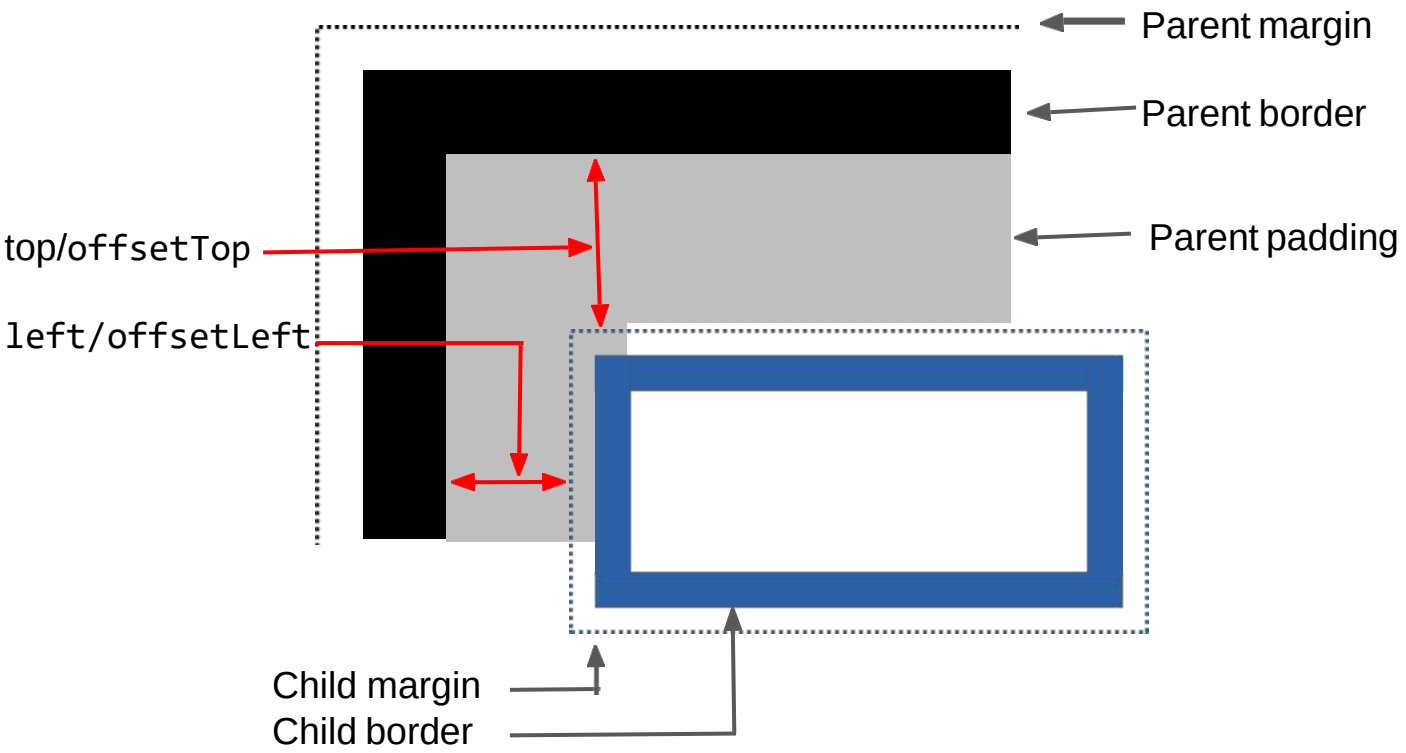
  "absolute" - the element no longer occupies space in the document flow.

- The origin inside an offsetParent (for positioning descendants) is just inside the upper-left corner of its border.

Positioning context

- Each element has an `offsetParent` (some ancestor element).
- When an element is positioned, coordinates such as `element.style.left` are relative to its `offsetParent`.
- Default `offsetParent` is the `<body>` element.
- Some elements define a new positioning context:
  - `position` CSS attribute is `absolute` (element is explicitly positioned)
  - `position` CSS attribute is `relative` (element is positioned automatically by the browser in the usual way)
  - This element will become the `offsetParent` for all its descendents (unless overridden by another positioning context)

Positioning Children



Element dimensions

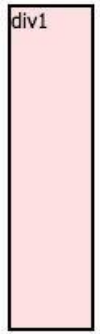
- Reading dimensions: `element.offsetWidth` and `element.offsetHeight`

  Include contents, padding, border, but not margin

- Updating dimensions: `element.style.width` and `element.style.height`

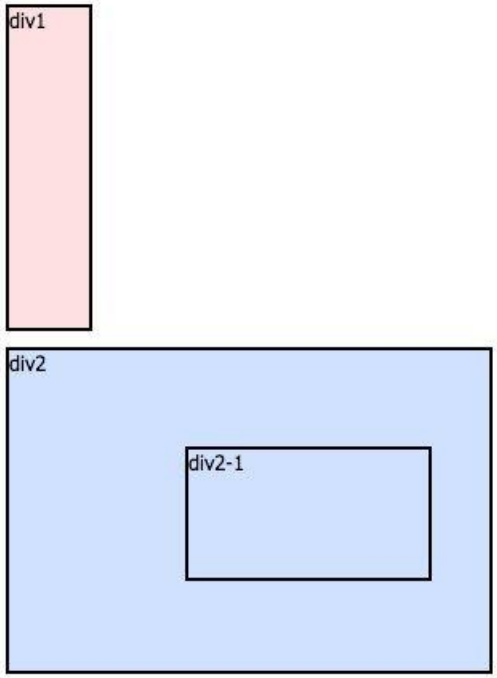
Positioning

```
<body>
    <div id="div1">
        <p>div1</p>
    </div>


#div1 {
   width: 50px;
   height: 200px;
   background: #ffe0e0;
}
```



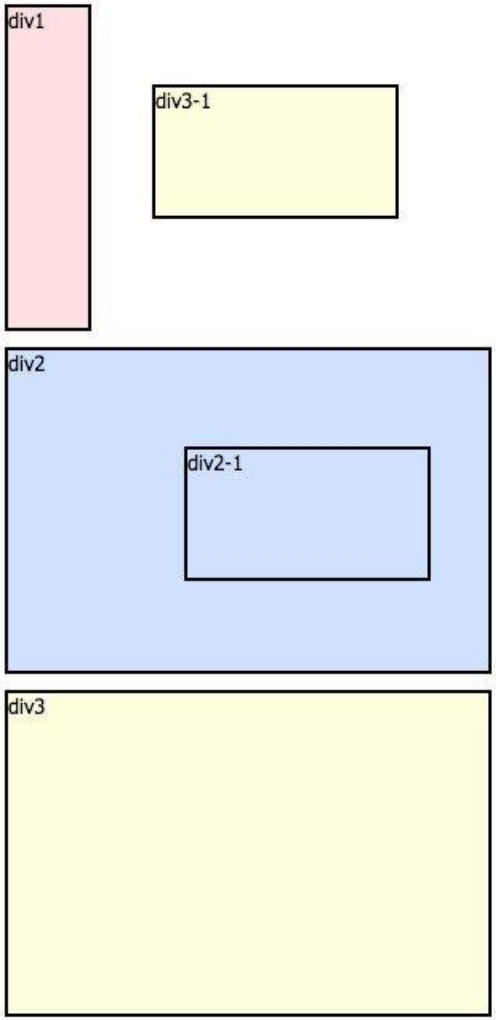
Positioning

```
    ...
    <div id="div2">
       <p>div2</p>
       <div id="div2-1">
          <p>div2-1</p>
       </div>
    </div>
#div2 {width: 300px; height:
200px; position: relative;
    background: #d0e0ff;}

#div2-1 {width: 150px; height:
80px; position: absolute;
   top: 50px; left: 100px;
background: #d0e0ff;}
```



Positioning

```
...
    <div id="div3">
       <p>div3</p>
       <div id="div3-1">
         <p>div3-1</p>
       </div>
    </div>
#div3 {width: 300px; height:
200px; background: #ffffe0;}

#div3-1 {width: 150px; height:
80px; position: absolute; top:
50px; left: 100px; background:
#ffffe0;}
```



DOM communicates to JavaScript with Events

Event types:

- Mouse-related: mouse movement, button click, enter/leave element
- Keyboard-related: down, up, press
- Focus-related: focus in, focus out (blur)
- Input field changed, Form submitted
- Timer events
- Miscellaneous:
  - Content of an element has changed
  - Page loaded/unloaded
  - Image loaded
  - Uncaught exception

Event handling

Creating an event handler: must specify 3 things:

- What happened: the event of interest.
- Where it happened: an element of interest.
- What to do: JavaScript to invoke when the event occurs on the element.

Specifying the JavaScript of an Event

- Option #1: in the HTML:

```
<div onclick="gotMouseClick('id42'); gotMouse=true;">...</div>
```

- Option #2: from Javascript using the DOM:

```
element.onclick = mouseClick;
    or
element.addEventListener("click", mouseClick);
```

- Example of the powerful listener/emitter pattern

Event object

- Event listener functions passed an `Event` object

  Typically sub-classed `MouseEvent, KeyboardEvent`, etc.

- Some Event properties:

  `type` - The name of the event (`'click', 'mouseDown', 'keyUp'`, …)

  `timeStamp` - The time that the event was created

  `currentTarget` - Element that listener was registered on

  `target` - Element that dispatched the event

`MouseEvent` and `KeyboardEvent`

- Some `MouseEvent` properties (prototype inherits from `Event`)

  `button` - mouse button that was pressed

  `pageX, pageY`: mouse position relative to the top-left corner of document

  `screenX, screenY`: mouse position in screen coordinates

- Some `KeyboardEvent` properties (prototype inherits from `Event`)

  `keyCode`: identifier for the keyboard key that was pressed

  Not necessarily an ASCII character!

  `charCode`: integer Unicode value corresponding to keypress, if there is one.

Draggable Rectangle - HTML/CSS

```
<style type="text/css">
  #div1 {
    position: absolute;
  }
</style>
...
<div id="div1" onmousedown="mouseDown(event);"
    onmousemove="mouseMove(event);"
    onmouseup="mouseUp(event);">Drag Me!</div>
```

Draggable Rectangle - JavaScript

```
var isMouseDown = false;   // Dragging?
var prevX, prevY;

function mouseDown(event) {
    prevX = event.pageX;
    prevY = event.pageY;
    isMouseDown = true;
}

function mouseUp(event) {
   isMouseDown = false;
}
```

```
function mouseMove(event) {
    if (!isMouseDown) {
      return;
    }
    var elem = document.getElementById("div1");
    elem.style.left = (elem.offsetLeft +
            (event.pageX - prevX)) + "px";
    elem.style.top = (elem.offsetTop +
            (event.pageY - prevY)) + "px";
    prevX = event.pageX;
    prevY = event.pageY;
}
```

Deciding which handler(s) are invoked for an event?

- Complicating factor: elements can contain or overlap other elements

  Suppose user clicks with the mouse on "xyz" in:

```
<body>
  <table>
    <tr>
      <td>xyz</td>
    </tr>
  </table>
</body>
```

  If I have handlers on the `td`, `tr`, `table`, and `body` elements which get called?

  - Sometimes only the innermost element should handle the event
  - Sometimes it's more convenient for an outer element to handle the event

Capturing and Bubbling Events

- Capture phase (or "trickle-down"):
  - Start at the outermost element and work down to the innermost nested element.
  - Each element can stop the capture, so that its children never see the event `event.stopPropagation()`

```
element.addEventListener(eventType, handler, true);
```

- Bubble phase - Most on handlers (e.g. `onclick`) use bubble, not `onfocus/blur`
  - Invoke handlers on the innermost nested element that dispatches the event (mostly right thing)
  - Then repeat on its parent, grandparent, etc. Any given element can stop the bubbling: `event.stopPropagation()`

```
element.addEventListener(eventType, handler, false);
```

- Handlers in the bubble phase more common than capture phase

Example: Timer Events

- Run `myfunc` once, 5 seconds from now:

  ```
  token = setTimeout(myFunc, 5*1000);
  ```

  Function is called in specified number of milliseconds

- Run `myfunc` every 50 milliseconds:

  ```
  token = setInterval(myfunc, 50);
  ```

- Cancel a timer:

  ```
  clearInterval(token);
  ```

- Used for animations, automatic page refreshes, etc.

Event Concurrency

- Events are serialized and processed one-at-a-time
- Event handling does not interleave with other Javascript execution.
  - Handlers run to completion
  - No multi-threading.
- Make reasoning about concurrency easier
  - Rarely need locks.
- Background processing is harder than with threads

Event-based programming is different

- Must wait for someone to invoke your code.
- Must return quickly from the handler (otherwise the application will lock up).
- Key is to maintain control through events: make sure you have declared enough handlers; last resort is a timer.
- Node.js uses event dispatching engine in JavaScript for server programming