[JavaScript 12]

Roi Yehoshua 2018

[DAN.IT]
EDUCATION

[What we learnt last time?]

- Working with DOM
- Inserting, moving, removing and cloning nodes
- Attributes
- Page Geometry

[Our targets for today]

- Browser events
- Attaching events
- Event bubbling
- Drag and Drop

Browser Events

- ➞ An **event** is a signal that something has happened
- ➞ Here’s a list of the most useful DOM events:

  →A full list can be found at https://www.w3schools.com/Jsref/dom_obj_event.asp

Event	Description
click	when the mouse clicks on an element (touchscreen devices generate it on a tap)
contextmenu	when the mouse right-clicks on an element
mouseover/mouseout	when the mouse cursor comes over / leaves an element
mousedown/mouseup	when the mouse button is pressed / released over an element
mousemove	when the mouse is moved
keydown/keyup	when the visitor presses and then releases the button
submit	when the visitor submits a <form>
focus	when the visitor focuses on an element, e.g. on an <input>
blur	when an element has lost focus
DOMContentLoaded	when the HTML is loaded and processed, DOM is fully built

Event Handlers

- To react on events we can assign a **handler** – a function that runs in case of an event
- Handlers is a way to run JavaScript code in case of user actions
- There are 3 ways to assign event handlers:
  - HTML attribute: onclick="..."
  - DOM property: elem.onclick = function
  - Methods: elem.addEventListener(event, handler[, phase])

[HTML-Attribute]

- A handler can be set in HTML with an attribute named **on<event>**
- For instance, to assign a click handler for an input, we can use **onclick**:

```
<input type="button" value="Click me" onclick="alert('Click!')"/>
```

  - On mouse click, the code inside onclick runs
  - Note that inside onclick we use single quotes, because the attribute itself is in double quotes
- An HTML-attribute is not a convenient place to write a lot of code, so we'd better create a JavaScript function and call it there
  - For example, the following function counts the number of clicks:

```
<script>
      let count = 0;
      function incrementCounter() {
            count++;
            alert("Number of clicks: " + count);
      }
</script>
<input type="button" value="Count!" onclick="incrementCounter()" />
```

[DOM Property]

- We can assign a handler using a DOM property **on<event>**
- For instance, elem.onclick:

```
<input id="elem" type="button" value="Click me"/>

<script>
    elem.onclick = function () {  alert("Thank
        you!");
    };
</script>
```

- If the handler is assigned using an HTML-attribute then the browser reads it, creates a new function from the attribute content and writes it to the DOM property
- So this way is actually the same as the previous one
- To remove a handler – assign elem.onclick = null

[DOM Property]

- → As there’s only one onclick property, we can’t assign more than one event handler
- → In the example below adding a handler with JS overwrites the existing handler:

```
<input id="elem" type="button" value="Click me" onclick="alert('Before')" />

<script>
    elem.onclick = function () {  alert('After');
    };
</script>
```

- → The value of **this** inside a handler is the element which has the handler on it
- → In the code below button shows its contents using this.innerHTML:

```
<button onclick="alert(this.innerHTML)">Click me</button>
```

[addEventListener]

- The previous methods don’t allow assigning multiple handlers to one event
- Another way of managing handlers which don’t suffer from this problem is by using the methods addEventListener() and removeEventListener()
- The syntax to add a handler:

```
element.addEventListener(event, handler[, phase]);
```

  - **event** – the event name, e.g. “click”
  - **handler** – the handler function
  - **phase** – an optional argument, the “phase” for the handler to work, will be discussed later

- To remove the handler, use removeEventListener:

```
element.removeEventListener(event, handler[, phase]);
```

  - To remove a handler we should pass exactly the same function as was assigned

addEventListener

- Multiple calls to addEventListener allow to add multiple handlers, like this:

```
<input id="btn" type="button" value="Click me" />

<script>
    function handler1() {  alert('Thanks!');
    }

    function handler2() {  alert('Thanks again!');
    }

    btn.addEventListener("click", handler1); // Thanks!
    btn.addEventListener("click", handler2); // Thanks again!
</script>
```

[Event Object]

- To properly handle an event we often need to know more about what’s happened
  - For example, in a “click” event what were the pointer coordinates? Or in “keypress”, which key was pressed?
- When an event happens, the browser creates an *event object*, puts details into it and passes it as an argument to the handler
- Here’s an example of getting mouse coordinates from the event object:

```
document.onclick = function (event) {
    alert(`Coordinates: (${event.clientX},${event.clientY})`);
};
```

- The event object is also accessible from HTML

```
<input type="button" value="Event type" onclick="alert(event.type)"/>
```

  - That’s possible because when the browser reads the attribute, it creates a handler like this: function(event) { alert(event.type) }

Event Object

Property	Description
type	The event type, e.g. “click”
currentTarget	The element that handled the event. That’s exactly the same as **this**, unless you bind **this** to something else.
target	The element that triggered the event
screenX / screenY	Coordinates of the mouse pointer relative to the screen
clientX / clientY	Coordinates of the mouse pointer relative to the window
pageX / pageY	Coordinates of the mouse pointer relative to the document
button	The mouse button that was pressed when the mouse event was triggered
key	The value of the key pressed by the user while taking into considerations the state of modifier keys such as the shiftKey
keyCode	the Unicode character code of the key that triggered the onkeypress event, or the Unicode key code of the key that triggered the onkeydown or onkeyup event
which	The same as **button** for mouse events and **keyCode** for keyboard events

→ Some useful properties of the event object:

[Exercise (12)]

→ Add JavaScript to the button to make <div id="text"> disappear when we click it

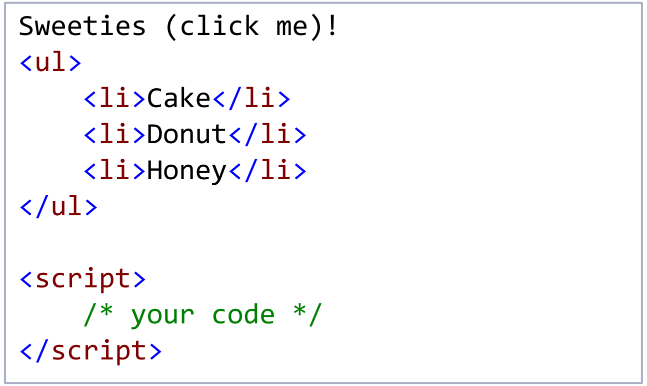
```
<input type="button" id="hider" value="Click to hide the text" />

<div id="text">Text</div>

<script>
    /* your code */
</script>
```

Exercise (13)

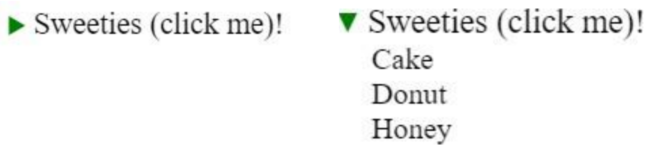
- Create a button that hides itself on click

Exercise (14)

- Create a menu that opens/collapses on click:



```
Sweeties (click me)!
<ul>
    <li>Cake</li>
    <li>Donut</li>
    <li>Honey</li>
</ul>

<script>
    /* your code */
</script>
```



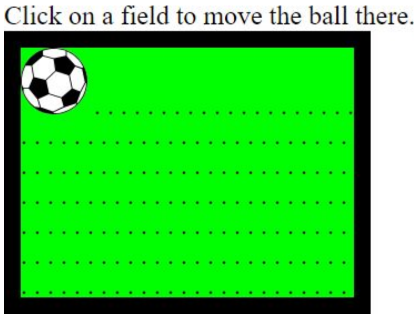
▶ Sweeties (click me)!

▼ Sweeties (click me)!
Cake
Donut
Honey

- The HTML/CSS of the source document should also be modified
- The arrow symbols are Unicode characters that can be copied from https://unicode-table.com/en/sets/arrows-symbols/

Exercise (15)

- Move the ball across a field when the ball is clicked

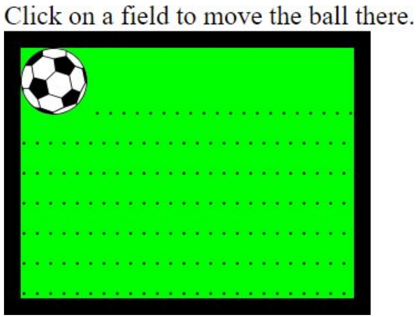


Click on a field to move the ball there.

- Requirements:
  - The ball center should come exactly under the pointer on click
  - The ball must not cross field boundaries
  - Use CSS-animation for showing the ball movement to the new location
  - The code should also work with different ball and field sizes, not be bound to any fixed values
- Use the HTML code on the next slide as a starter page

Exercise (15)

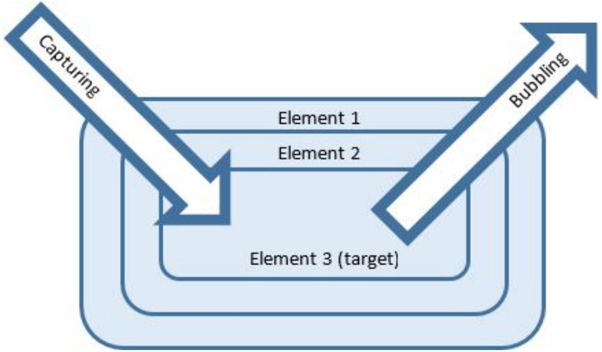
```
<html>
<head>
    <style>
        #field {
            width: 200px;  height: 150px;
            border: 10px solid black;  background-color: #00FF00;
            overflow: hidden;
        }
    </style>
</head>
<body>
    Click on a field to move the ball there.

    <div id="field">
        <img  src="https://en.js.cx/clipart/ball.svg" id="ball">  . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . . . . . . . . . . . . . . . . . . . . . . . . . .
    </div>
</body>
</html>
```



Click on a field to move the ball there.

Bubbling and Capturing

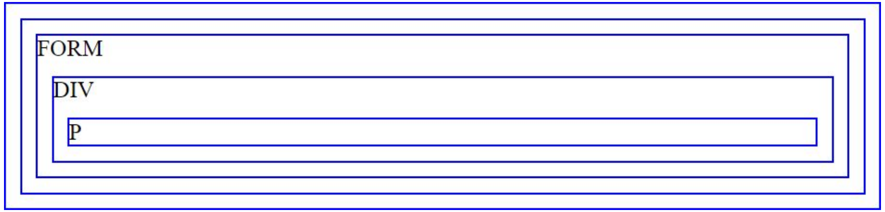
- Event bubbling and capturing are two ways of event propagation in the HTML DOM
- When an event occurs in an element inside another element, and both elements have registered a handle for that event:
  - With **bubbling**, the event is first captured and handled by the innermost element and then propagated to outer elements
  - With **capturing**, the event is first captured by the outermost element and propagated to the inner elements

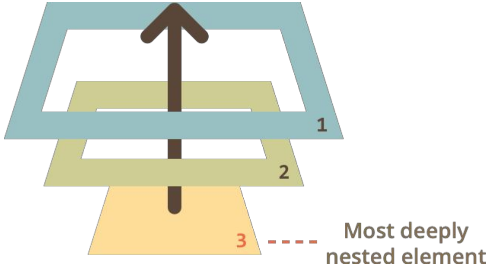


Bubbling

- Let’s say, we have 3 nested elements FORM > DIV > P, each one with a handler:

```
<style>
    * {
        margin: 10px;
        border: 1px solid blue;
    }
</style>
<form onclick="alert('form')">FORM
    <div onclick="alert('div')">DIV
        <p onclick="alert('p')">P</p>
    </div>
</form>
```





- A click on the inner <p> first runs onclick:
  - On that <p>
  - Then on the outer <div>
  - Then on the outer <form>
  - And so on upwards till the document object

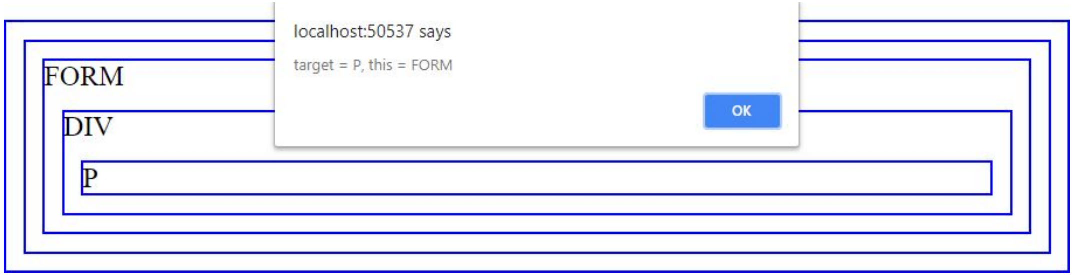
[event.target]

- A handler on a parent element is able to know where it actually happened
- The most deeply nested element that caused the event is called a *target* element, accessible as **event.target**
- Note the differences from **this** (=event.currentTarget):
  - event.target – is the “target” element that initiated the event, it doesn’t change through the bubbling process
  - this – is the “current” element, the one that has a currently running handler on it
- For instance, if we have a single handler **form.onclick**, then it can “catch” all clicks inside the form
- In form.onclick handler:
  - this (=event.currentTarget) is the <form> element, because the handler runs on it
  - event.target is the concrete element inside the form that actually was clicked

event.target

```
<form id="form">FORM
    <div>DIV
        <p>P</p>
    </div>
</form>

<script>
    form.onclick = function (event) {
        alert("target = " + event.target.tagName + ", this = " + this.tagName);
    };
</script>
```

- → A click on the inner <p> shows the following message:

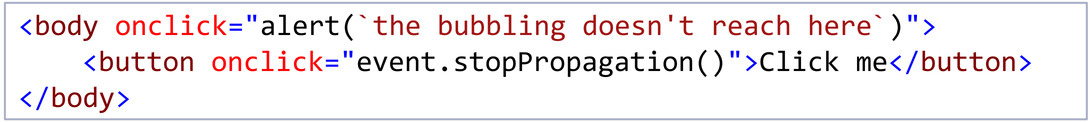


localhost:50537 says

target = P, this = FORM

Stopping Bubbling

- → A bubbling event goes from the target element straight up
- → Normally it goes upwards till <html>, and then to document object, and some events even reach window, calling all handlers on the path
- → But any handler may decide to stop the bubbling by calling **event.stopPropagation()**
- → For instance, here body.onclick doesn’t work if you click on <button>:

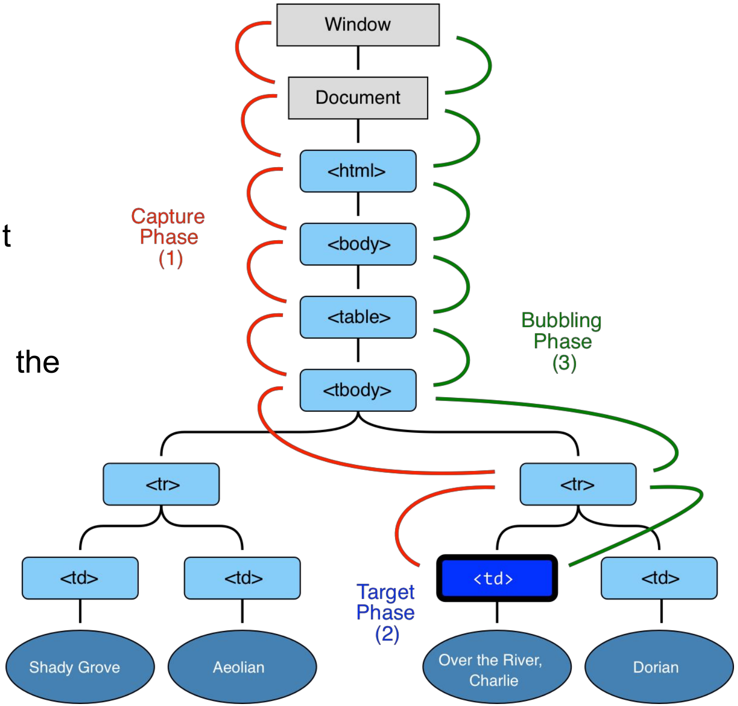


```
<body onclick="alert(`the bubbling doesn't reach here`)">
    <button onclick="event.stopPropagation()">Click me</button>
</body>
```

- → Bubbling is convenient. Don’t stop it without a real need.

Event Propagation Phases

- ➞ There are 3 phases of event propagation:
  - → **Capturing phase** – the event goes down to the element
  - → **Target phase** – the event reached the target element
  - → **Bubbling phase** – the event bubbles up from the element
- ➞ For example, when clicking a <td>:
  - → the event first goes through the ancestors chain down to the element (capturing)
  - → it reaches the target,
  - → then it goes up (bubbles), calling handlers on its way

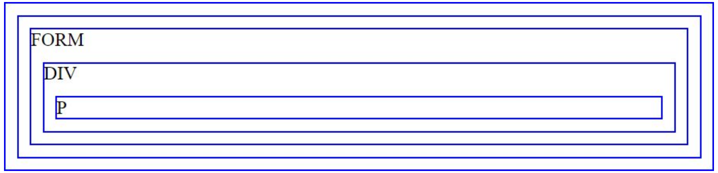


[Capturing]

- The capturing phase is rarely used. Normally it is invisible to us.
- Handlers added using on<event>-property, HTML attributes, or addEventListener(event, handler), don't know anything about capturing
  - They only run on the 2nd and 3rd phases
- To catch an event on the capturing phase, we need to set the 3rd argument of addEventListener to **true**
- There are two possible values for that optional last argument:
  - If it's false (default), then the handler is set on the bubbling phase
  - If it's true, then the handler is set on the capturing phase

Capturing

```
<form>FORM
     <div>DIV
          <p>P</p>
     </div>
</form>
<script>
     for (let elem of document.querySelectorAll('*')) {
          elem.addEventListener("click", e => alert(`Capturing: ${elem.tagName}`), true);
          elem.addEventListener("click", e => alert(`Bubbling: ${elem.tagName}`));
     }
</script>
```



- If you click on <p>, then the sequence is:
  - HTML → BODY → FORM → DIV → P (capturing phase, the first listener), and then:
  - P → DIV → FORM → BODY → HTML (bubbling phase, the second listener)

[Summary]

- The event handling process:
  - When an event happens – the most nested element where it happens gets labeled as the “target element” (event.target)
  - Then the event first moves from the document root down the event.target, calling handlers assigned with addEventListener(...., true) on the way
  - Then the event moves from event.target up to the root, calling handlers assigned using on<event> and addEventListener without the 3rd argument or with the 3rd argument false
- Each handler can access event object properties:
  - event.target – the deepest element that originated the event
  - event.currentTarget (=this) – the current element that handles the event (the one that has the handler on it)
  - event.eventPhase – the current phase (capturing=1, target=2, bubbling=3)

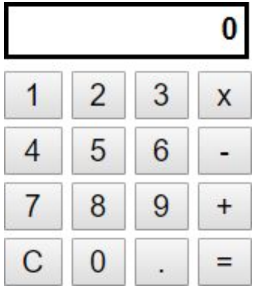
Event Delegation

- Capturing and bubbling allow us to implement one of most powerful event handling patterns called **event delegation**
- The idea is that if we have a lot of elements handled in a similar way, then instead of assigning a handler to each of them – we put a single handler on their common ancestor
- In the handler we get **event.target**, see where the event actually happened and handle it

Event Delegation

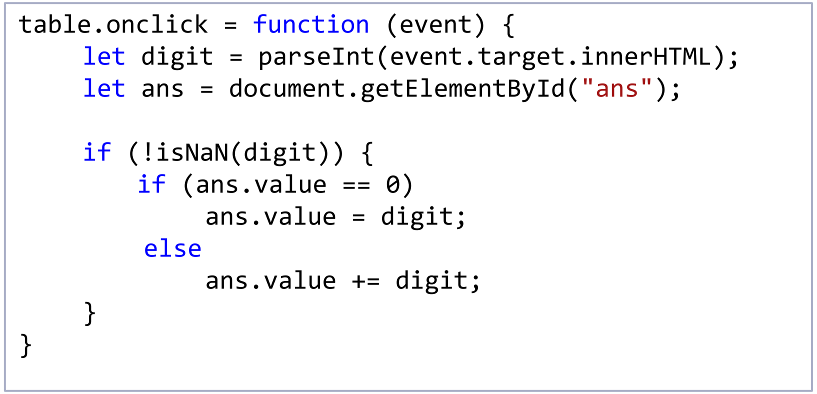
- → For example, let’s say we want to build a simple calculator
- → The HTML is like this:

```
              <div id="calculator">
    <input id="ans" type="text" value="0" />
    <table id="table">
        <tr>
            <td><button>1</button></td>
            <td><button>2</button></td>
            <td><button>3</button></td>
            <td><button>x</button></td>
        </tr>
        <tr> … </tr>
        <tr> … </tr>
        <tr> … </tr>
    </table>
</div>
```

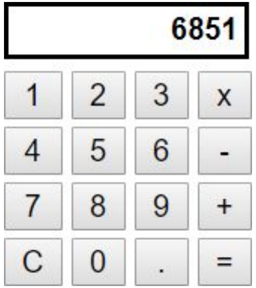


Event Delegation

- ➞ Instead of assigning an onclick handler to each <button> – we’ll setup the “catch-all” handler on the <table> element
- ➞ It will use event.target to get the clicked element and perform the relevant action:



```
table.onclick = function (event) {
    let digit = parseInt(event.target.innerHTML);
    let ans = document.getElementById("ans");

    if (!isNaN(digit)) {
        if (ans.value == 0)
             ans.value = digit;
        else
             ans.value += digit;
    }
}
```



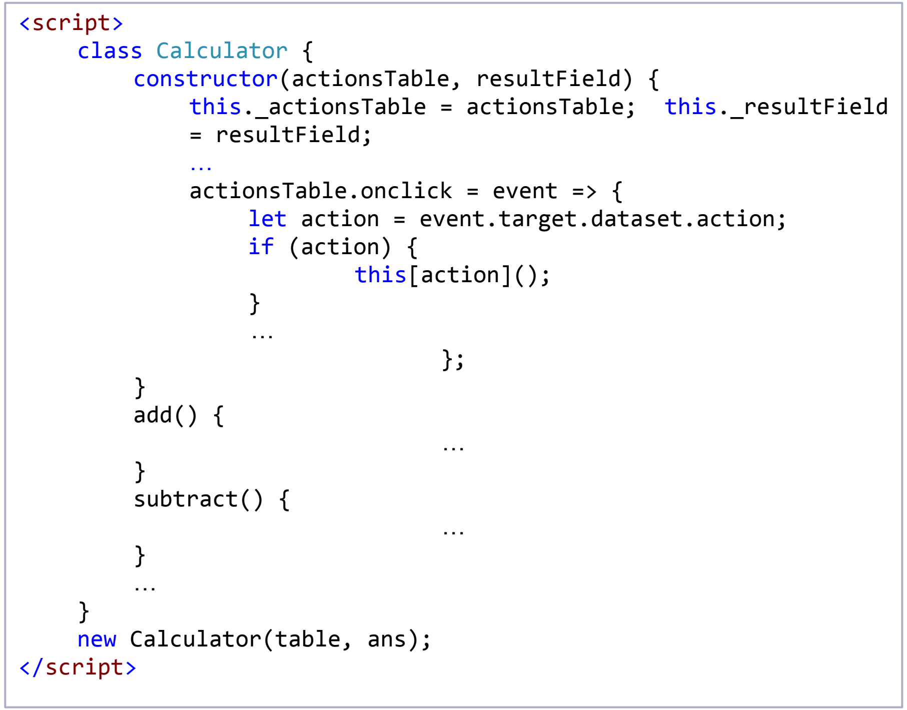
[Event Delegation]

- ➞ We’ve used a single handler for similar actions on many elements
- ➞ But we can also use a single handler as an entry point for many different actions
- ➞ For instance, we want to handle the calculator operators +, -, *, and so on, with a single handler
- ➞ We can create an object with methods add(), subtract(), multiply(), etc.
- ➞ Then we can add data-action attributes for the buttons with the method to call:

```
<button data-action="multiply">x</button>
<button data-action="subtract">-</button>
<button data-action="add">+</button>
```

- ➞ The event handler reads the attribute and executes the method

Event Delegation



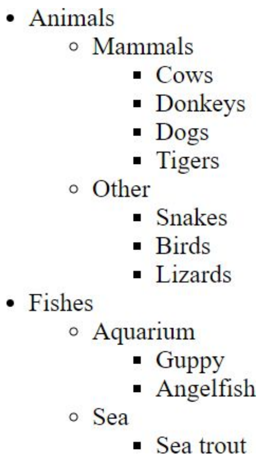
```
<script>
    class Calculator {
        constructor(actionsTable, resultField) {
            this._actionsTable = actionsTable;  this._resultField
            = resultField;
            …
            actionsTable.onclick = event => {
                let action = event.target.dataset.action;
                if (action) {
                        this[action]();
                }
                …
                            };
        }
        add() {
                            …
        }
        subtract() {
                            …
        }
        …
    }
    new Calculator(table, ans);
</script>
```

What the delegation gives us here?

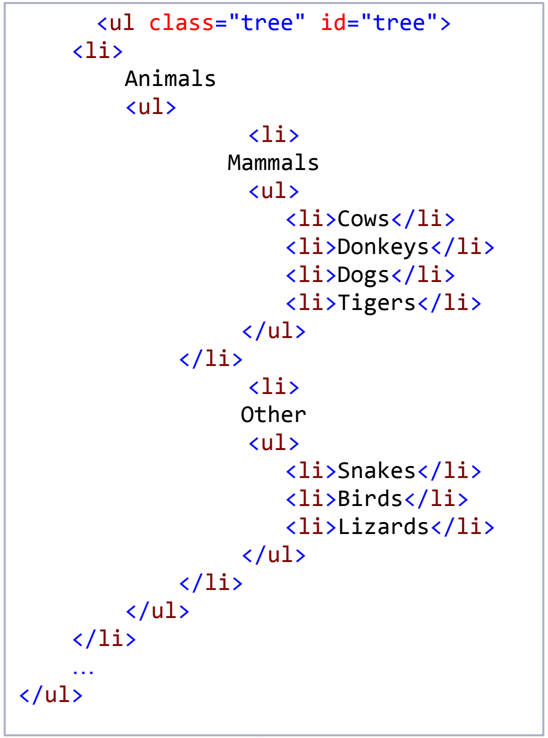
- We don’t need to write the code to assign a handler to each button. Just make a method and put it in the markup.
- The HTML structure is flexible, we can add/remove buttons at any time.

Exercise (16)

- Create a tree that shows/hides node children on click:



- Animals
  - Mammals
    - Cows
    - Donkeys
    - Dogs
    - Tigers
  - Other
    - Snakes
    - Birds
    - Lizards
- Fishes
  - Aquarium
    - Guppy
    - Angelfish
  - Sea
    - Sea trout



```
      <ul class="tree" id="tree">
    <li>
        Animals
        <ul>
                 <li>
                Mammals
                 <ul>
                    <li>Cows</li>
                    <li>Donkeys</li>
                    <li>Dogs</li>
                    <li>Tigers</li>
                 </ul>
            </li>
                 <li>
                 Other
                 <ul>
                    <li>Snakes</li>
                    <li>Birds</li>
                    <li>Lizards</li>
                 </ul>
            </li>
        </ul>
    </li>
    ...
</ul>
```

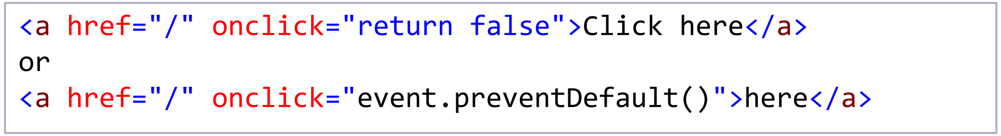
- Requirements:
  - → Only one event handler (use delegation)
  - → A click outside a node title should not do anything

Browser Default Actions

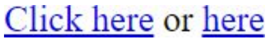
- Many events automatically lead to browser actions
- For instance:
  - A click on a link – initiates going to its URL
  - A click on submit button inside a form – initiates its submission to the server
  - Pressing a mouse button over a text and moving it – selects the text
- If we handle an event in JavaScript, often we don’t want the browser action
- There are two ways to tell the browser we don’t want it to act:
  - The main way is to use the method **event.preventDefault()** of the event object
  - If the handler is assigned using on<event> (not by addEventListener), then we can just return **false** from it

Preventing Browser Actions

→ In the example below a click to links don’t lead to URL change:



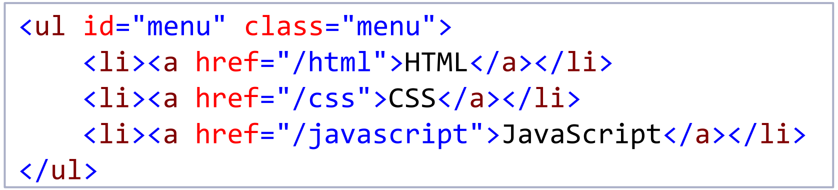
```
<a href="/" onclick="return false">Click here</a>
or
<a href="/" onclick="event.preventDefault()">here</a>
```



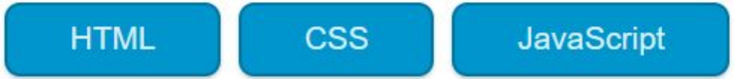
Click here or here

[Preventing Browser Actions]

- ➞ Consider a site menu, like this:



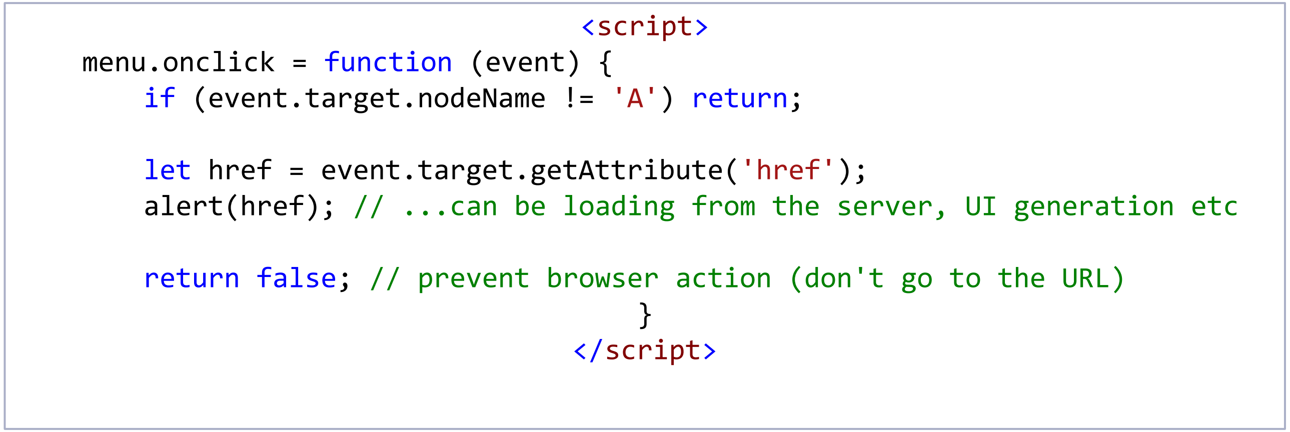
```html
<ul id="menu" class="menu">
    <li><a href="/html">HTML</a></li>
    <li><a href="/css">CSS</a></li>
    <li><a href="/javascript">JavaScript</a></li>
</ul>
```



- ➞ Menu items are links <a>, not buttons. There are several benefits, for instance:
  - → Many people like to use “right click” – “open in a new window”. If we use <button> or <span>, that doesn’t work.
  - → Search engines follow <a href="..."> links while indexing.
- ➞ So we use <a> in the markup, but normally we intend to handle clicks in JavaScript
- ➞ So we should prevent the default browser action.

Preventing Browser Actions

→ Consider a site menu, like this:

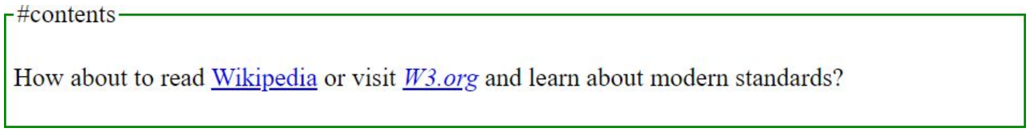


```
                               <script>
menu.onclick = function (event) {
    if (event.target.nodeName != 'A') return;

    let href = event.target.getAttribute('href');
    alert(href); // ...can be loading from the server, UI generation etc

    return false; // prevent browser action (don't go to the URL)
                                   }
                              </script>
```

→ If we omit return false, then after our code executes the browser will do its “default action” – following to the URL in href.

Exercise (17)

- Make all links inside the element with id="contents" ask the user if he really wants to leave. And if he doesn’t then don’t follow.



#contents

How about to read Wikipedia or visit *W3.org* and learn about modern standards?

- Note the following:
  - The HTML inside the element may be loaded or regenerated dynamically at any time, so we can’t find all links and put handlers on them. Use event delegation.
  - The content may have nested tags, inside links too, like <a href=".."><i>...</i></a>.
- Start with the code on the following slide

Exercise (17)

```
<head>
    <style>
        #contents {
            padding: 5px;
            border: 1px green solid;
        }
    </style>
</head>
<body>
    <fieldset id="contents">
        <legend>#contents</legend>
        <p>
            How about to read <a href="http://wikipedia.org">Wikipedia</a>  or
visit <a href="http://w3.org"><i>W3.org</i></a> and learn about modern
standards?
        </p>
    </fieldset>

    <script>
        // Your code here
    </script>
</body>
```

[Mouse Events]

- Mouse events are not only invoked by mouse devices, but are also emulated on touch devices, to make them compatible
- We can split mouse events into two categories:
  - Simple events:

Event	Description
mousedown/mouseup	Mouse button is clicked/released over an element
mouseover/mouseout	Mouse pointer comes over/out from an element
mousemove	Every mouse move over an element triggers that event

  - Complex events (which are made of simpler ones):

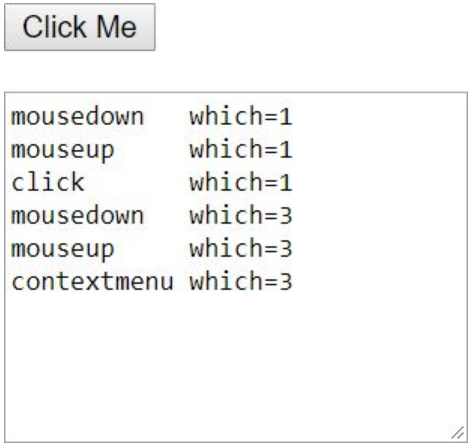
Event	Description
click	Triggers after mousedown and then mouseup over the same element if the left mouse button was used
contextmenu	Triggers after mousedown if the right mouse button was used
dblclick	Triggers after a double click over an element

[DAN.IT] EDUCATION

[Events Order]

- An action may trigger multiple events
- For instance, a click first triggers **mousedown**, when the button is pressed, then **mouseup** and **click** when it’s released
- Thus, the event handlers are called in the order mousedown → mouseup → click

```
<button onmousedown="logMouse(event)" onmouseup="logMouse(event)"
onclick="logMouse(event)" oncontextmenu="logMouse(event)"
ondblclick="logMouse(event)">Click Me</button><br/><br/>

<textarea id="logArea" style="font-size: 12px; height:150px;
width:200px"></textarea>

<script>
    function logMouse(event) {  let type = event.type;
        while (type.length < 11) type += ' ';
        logArea.value += `${type} which=${event.which}\n`;
                                  }
</script>
```



```
mousedown   which=1
mouseup     which=1
click       which=1
mousedown   which=3
mouseup     which=3
contextmenu which=3
```

Getting the Button: which

- Click-related events have the **which** property, which gives the exact mouse button
  - Only relevant for mousedown and mouseup events
  - Because click happens only on left-click, and contextmenu happens only on right-click
- There are three possible values:
  - event.which == 1 – the left button
  - event.which == 2 – the middle button
  - event.which == 3 – the right button
- The middle button is somewhat exotic right now and is very rarely used

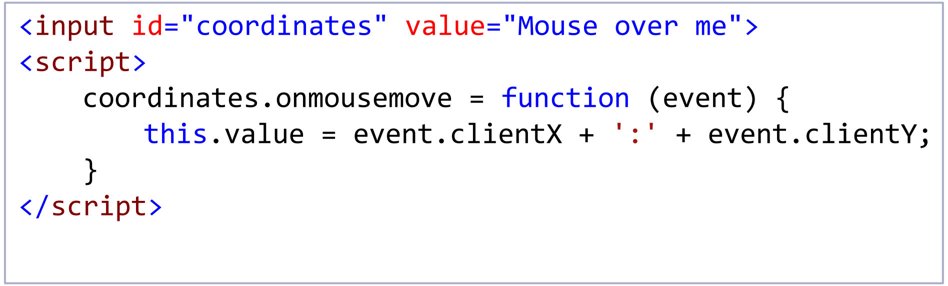
Modifiers

- All mouse events include the information about pressed modifier keys
- The properties are:
  - shiftKey
  - altKey
  - ctrlKey
  - metaKey (Cmd for Mac)
- For instance, the button below only works on Alt+Shift+click:

```
<button id="button">Alt+Shift+Click on me!</button>
<script>
    button.onclick = function (event) {
        if (event.altKey && event.shiftKey) {
            alert("Hooray!");
        }
    }
</script>
```

[Coordinates: clientX/Y, pageX/Y]

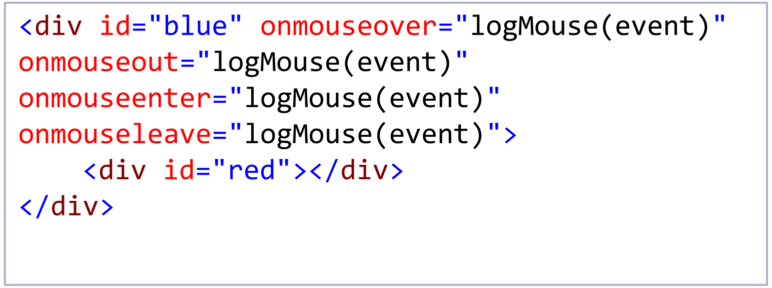
- All mouse events have coordinates in two flavours:
  - Window-relative: clientX and clientY
  - Document-relative: pageX and pageY
- Move the mouse over the input field to see clientX/clientY:



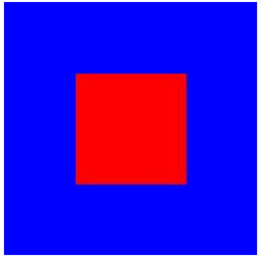
```
<input id="coordinates" value="Mouse over me">
<script>
    coordinates.onmousemove = function (event) {
        this.value = event.clientX + ':' + event.clientY;
    }
</script>
```

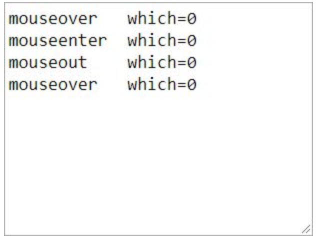
Events mouseenter and mouseleave

- ➞ Events mouseenter/mouseleave are like mouseover/mouseout
- ➞ They also trigger when the mouse pointer enters/leaves the element
- ➞ However, there are two differences:
  - → Transitions inside the element are not counted
  - → Events mouseenter/mouseleave do not bubble



```
<div id="blue" onmouseover="logMouse(event)"
onmouseout="logMouse(event)"
onmouseenter="logMouse(event)"
onmouseleave="logMouse(event)">
    <div id="red"></div>
</div>
```



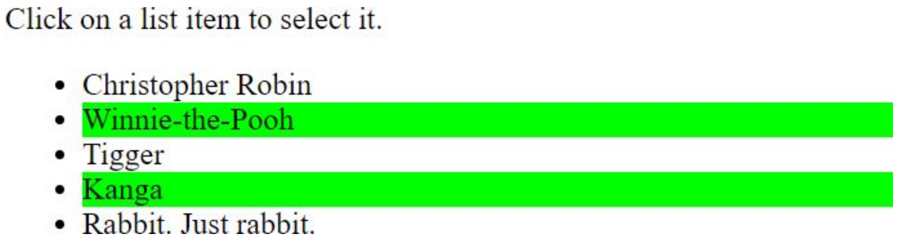


```
mouseover   which=0
mouseenter  which=0
mouseout    which=0
mouseover   which=0
```

- ➞ The mouseenter/mouseleave trigger only on entering and leaving the blue <div>
- ➞ The mouseleave event only triggers when the cursor leaves it

Exercise (18)

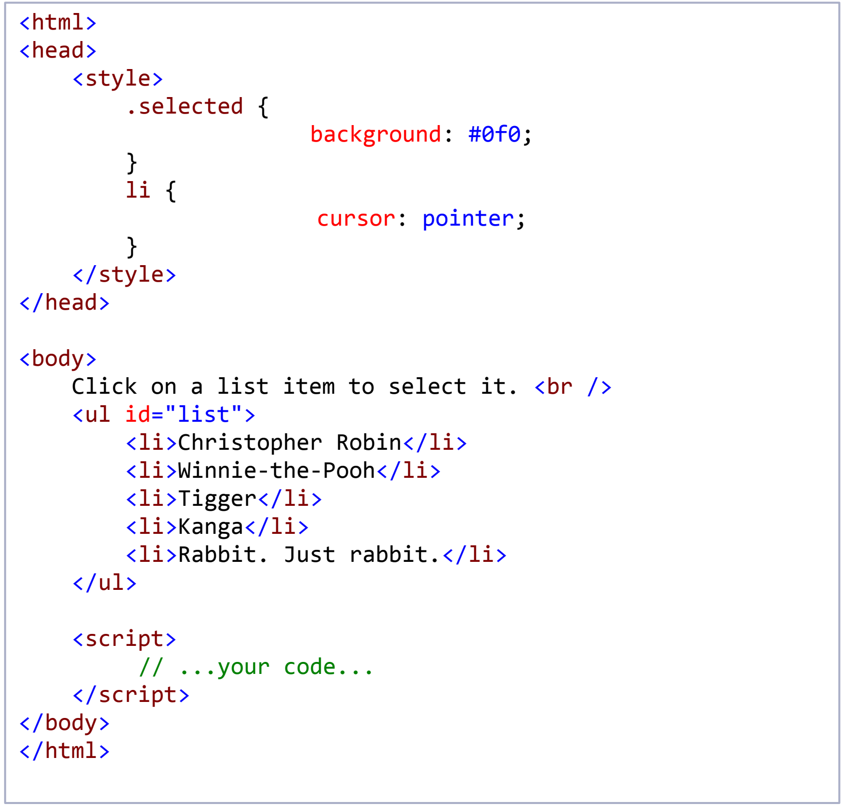
- → Create a list where elements are selectable, like in file-managers
- → A click on a list element selects only that element (adds the class .selected), and deselects all others
- → If a click is made with Ctrl (Cmd for Mac), then the selection is toggled on the element, but other elements are not modified
- → Start with the HTML page on the next slide



Click on a list item to select it.

- Christopher Robin
- Winnie-the-Pooh
- Tigger
- Kanga
- Rabbit. Just rabbit.

Exercise (18)



```
<html>
<head>
    <style>
        .selected {
                      background: #0f0;
        }
        li {
                       cursor: pointer;
        }
    </style>
</head>

<body>
    Click on a list item to select it. <br />
    <ul id="list">
        <li>Christopher Robin</li>
        <li>Winnie-the-Pooh</li>
        <li>Tigger</li>
        <li>Kanga</li>
        <li>Rabbit. Just rabbit.</li>
    </ul>

    <script>
         // ...your code...
    </script>
</body>
</html>
```

[Drag and Drop]

- Drag and drop is a very common feature: it allows you to take an object, drag it and drop it in another location
- This provides a simple way to do many things, from copying and moving files to ordering (drop into cart)
- The basic Drag’n’Drop algorithm looks like this:
  - Catch mousedown on a draggable element
  - Prepare the element to moving (maybe create a copy of it or whatever)
  - Then on mousemove move it by changing left/top and position:absolute
  - On mouseup (button release) – perform all actions related to a finished Drag’n’Drop

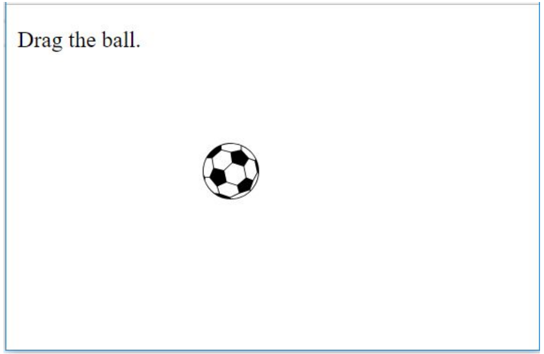
Drag and Drop Example

```
<p>Drag the ball.</p>
<img id="ball" src="https://js.cx/clipart/ball.svg" style="cursor: pointer"/>

<script>
    let ball = document.getElementById("ball");

    ball.onmousedown  =   function (event) {      // start the process
         // prepare to moving: make absolute and on top by z-index
         ball.style.position = "absolute";  ball.style.zIndex = 1000;

         // move the ball out of any current parents directly into body
         // to make it positioned relative to the body
         document.body.append(ball);

         // put the absolute positioned ball under the cursor  moveAt(event.pageX, event.pageY);

         // move the ball onmousemove
         // we track mousemove on document, not on ball, because mousemove doesn't trigger
         // for every pixel, thus the cursor may leave the ball's area on a swift move
         document.addEventListener("mousemove", onMouseMove);
```

[DAN.IT]
EDUCATION

Drag and Drop Example

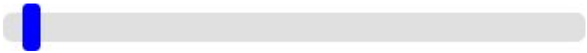
```
        // drop the ball, remove unneeded handlers  ball.onmouseup =
        function (event) {
            document.removeEventListener("mousemove", onMouseMove);
            ball.onmouseup = null;
        }

        function onMouseMove(event) {  moveAt(event.pageX, event.pageY);
        }

        // center the ball at (pageX, pageY) coordinates  function
        moveAt(pageX, pageY) {
            ball.style.left = pageX - ball.offsetWidth / 2 + "px";
            ball.style.top = pageY - ball.offsetHeight / 2 + "px";
        }
    }

    // Disable the browser's default behavior for drag'n'drop
    ball.ondragstart = function (event) {
        event.preventDefault();
    }
                           </script>
```



Drag the ball.

Exercise (19)

- Create a slider:



- Drag the blue thumb with the mouse and move it
- Important details:
  - When the mouse button is pressed, during the dragging the mouse may go over or below the slider. The slider will still work (convenient for the user).
  - If the mouse moves very fast to the left or to the right, the thumb should stop exactly at the edge.
- Start with the HTML on the next slide

Exercise (19)

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8" />
    <style>
        .slider {
            border-radius: 5px;  background: #E0E0E0;
            background: linear-gradient(left top, #E0E0E0, #EEEEEE);
            width: 310px;
            height: 15px;  margin: 5px;
        }

        .thumb {
            width: 10px;  height: 25px;  border-radius: 3px;
            position: relative;  left: 10px;
            top: -5px;  background: blue;  cursor: pointer;
        }
                               </style>
</head>
```

```
<body>
    <div id="slider" class="slider">
            <div class="thumb"></div>
    </div>

    <script>
               // ...your code...
    </script>
</body>
</html>
```

Control questions

1. What is Event?
2. How can we attach an event on the page?
3. What is event bubbling?
4. What is Drag and Drop and how can we create one?

[Materials]

Core materials:

https://learn.javascript.ru/introduction-browser-events
https://learn.javascript.ru/event-details

Additional materials:

https://developer.mozilla.org/ru/docs/Web/Events
https://learn.javascript.ru/event-bubbling

Video materials:

https://www.youtube.com/watch?v=8cV4ZvHXQL4