[Lesson 12-14]

Roi Yehoshua 2018

[DAN.IT]
EDUCATION

[What we learnt last time?]

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

[DAN.IT]
EDUCATION

[Our targets for today]

- Browser events
- Attaching events to DOM nodes
- Keyboard events
- Page events
- Input events
- Form events
- Bubbling and capturing events
- Event delegation
- Preventing Browser Actions

[DAN.IT]
EDUCATION

[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

[DAN.IT]
EDUCATION

[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])

[DAN.IT] EDUCATION

[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!')"/>
```

Click me

  - 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()" />
```

Count!

[DAN.IT] EDUCATION

[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

[DAN.IT] EDUCATION

[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>
```

[DAN.IT] EDUCATION

[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

[DAN.IT] EDUCATION

[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>
```

[DAN.IT] EDUCATION

[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) }

[DAN.IT] EDUCATION

[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:

[DAN.IT] EDUCATION

[Exercise (1)]

→ Add JavaScript to the button to make 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>
```

[DAN.IT] EDUCATION

[Exercise (2)]

→ Create a button that hides itself on click

Click to hide

[DAN.IT] EDUCATION

[Exercise (3)]

- 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)!

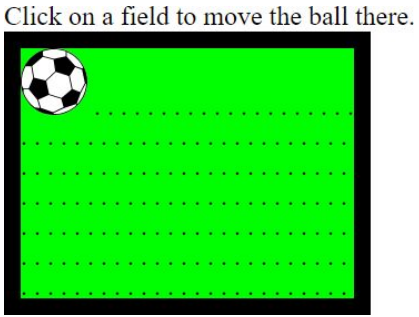
▼ Sweeties (click me)!
Cake
Donut
Honey

- 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/

[DAN.IT] EDUCATION

[Exercise (4)]

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



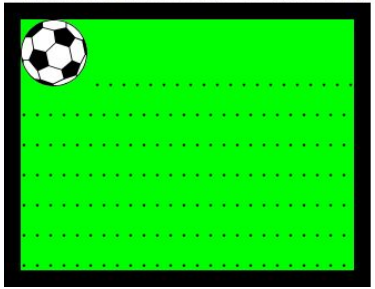
- Requirements:
  - The ball center should come exactly under the pointer on click
  - The ball must not cross field boundaries
  - Use CSS-animation for showing the ball movement to the new location
  - The code should also work with different ball and field sizes, not be bound to any fixed values
-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

[DAN.IT] EDUCATION

[Exercise (4)]

```
<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.



[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>
```

Click Me

```
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

[DAN.IT] EDUCATION

[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>
```

Alt+Shift+Click on me!

[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:

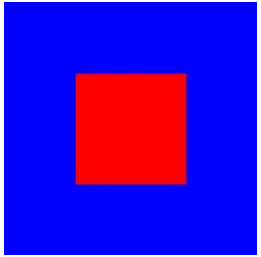
```
<input id="coordinates" value="Mouse over me">
<script>
    coordinates.onmousemove = function (event) {
        this.value = event.clientX + ':' +
        event.clientY;
    }
</script>
```

86:274

[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
- The mouseleave event only triggers when the cursor leaves it

[DAN.IT] EDUCATION

[Exercise (5)]

- 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.

[DAN.IT] EDUCATION

[Exercise (5)]

```html
<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>
```

[DAN.IT] EDUCATION

[Keyboard Events]

- Keyboard events should be used when we want to handle keyboard actions
- Note that on modern devices there are other ways to “input something”
  - For instance, people use speech recognition (especially on mobile devices) or copy/paste with the mouse
- So if we want to track any input into an <input> field, then keyboard events are not enough
- There’s another event named **input** to handle changes of an <input> field, by any means, that will be discussed later

[DAN.IT] EDUCATION

[Keyboard Events]

- **keydown** happens when a key is pressed down, and then **keyup** – when it's released
  - In the past, there was also a keypress event, but now is considered deprecated
- The **key** property of the event object allows to get the character
  - The value of event.key can change depending on the language or CapsLock enabled
- The **code** property of the event object allows to get the "physical key code"
- For instance, the same key Z can be pressed with or without Shift:

Key	event.key	event.code
Z	z (lowercase)	KeyZ
Shift+Z	Z (uppercase)	KeyZ

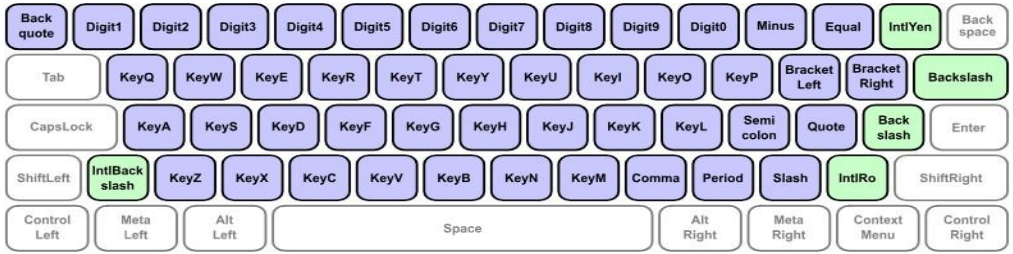
- For non-character keys, key usually has the same value as code, for example:

Key	event.key	event.code
F1	F1	F1
Shift	Shift	ShiftRight or ShiftLeft

[DAN.IT] EDUCATION

[Key Codes]

- Every key has a code that depends on its location on the keyboard
- The key codes are described in the UI Events code specification
- For example:
  - Letter keys have codes "Key<letter>": "KeyA", "KeyB" etc.
  - Digit keys have codes: "Digit<number>": "Digit0", "Digit1" etc.
  - Special keys are coded by their names: "Enter", "Backspace", "Tab" etc.



[Keyboard Events Example]

- For example, let's say we want to implement an "Undo" action when the user presses Ctrl+Z (or Cmd+Z) for Mac
- We can set a listener on **keydown** and check event.code for the key pressed
  - we don't want event.key here, since the value of event.key can change depending on the language or CapsLock enabled

```
document.addEventListener("keydown", function (event) {
    if (event.code == "KeyZ" && (event.ctrlKey || event.metaKey))
    {
        alert("Undo!");
    }
});
```

[Default Actions]

- Default actions vary, as there are many possible things that may be initiated by the keyboard, for example:
  - A character appears on the screen (the most obvious outcome)
  - A character is deleted (Delete key)
  - The page is scrolled (PageDown key)
  - The browser opens the “Save Page” dialog (Ctrl+S)
- Preventing the default action on keydown can cancel most of them, with the exception of OS-based special keys
- For instance, on Windows Alt+F4 closes the current browser window, and there’s no way to stop it by preventing the default action in JavaScript

[DAN.IT] EDUCATION

[Default Actions]

→ For instance, the <input> below expects a phone number, so it doesn't accept keys except digits, +, -, or ():

```
<input type="tel" id="phone" placeholder="Phone, please">
<script>
    phone.onkeydown = function (event) {
        if (!checkPhoneKey(event.key))
            event.preventDefault();
    }
    function checkPhoneKey(key) {
        return (key >= '0' && key <= '9') || key == '+' || key == '-' ||  key
            == '(' || key == ')';
    }
</script>
```

Phone, please

→ Note that special keys like Backspace, ⇦, ⇨, Ctrl+V don't work in the input. We can relax the filter checkPhoneKey() to allow these keys as well if we want.

→ We can still enter anything by using a mouse and right-click + Paste. So the filter is not 100% reliable. Alternatively, we could track the input event – it triggers after any modification.

[DAN.IT] EDUCATION

[Exercise (6)]

- Create a function **runOnKeys**(func, code1, code2, ... code_n) that runs func on simultaneous pressing of keys with codes code1, code2, …, code_n.
- For instance, the code below shows alert when "Q" and "W" are pressed together (in any language, with or without CapsLock)

```
runOnKeys(
    () =>
    alert("Hello!"),
    "KeyQ",
    "KeyW"
);
```

[DAN.IT] EDUCATION

[Page LifeCycle]

- The lifecycle of an HTML page has three important events:
  - **DOMContentLoaded** – the browser fully loaded HTML, and the DOM tree is built, but external resources like pictures <img> and stylesheets may be not yet loaded
  - **load** – the browser loaded all resources (images, styles etc)
  - **beforeunload/unload** – when the user is leaving the page

[DAN.IT] EDUCATION

[DOMContentLoaded]

- The DOMContentLoaded event happens on the document object
- We must use addEventListener() to catch it:

```
<script>
    function ready() {
        alert("DOM is ready");

        // image is not yet loaded (unless was cached), so the size is 0x0
        alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
    }

    document.addEventListener("DOMContentLoaded", ready);
</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

→ In the example the DOMContentLoaded handler runs when the document is loaded, not waiting for the page load. So alert shows a size of zero.

[DAN.IT] EDUCATION

[DOMContentLoaded and Scripts]

- When the browser initially loads HTML and comes across a <script>...</script> in the text, it must execute the script before continuing building the DOM
- So DOMContentLoaded may only happen after all such scripts are executed
- External scripts (with src) also put DOM building to pause while the script is loading
- However, external scripts with **async** or **defer** attributes tell the browser to continue processing without waiting for the scripts, and run them when they finish loading
  - So the user can see the page before scripts finish loading, good for performance

	async	**defer**
Order	Scripts with async execute in the load-first order. Their document order doesn't matter – which loads first runs first.	Scripts with defer always execute in the document order (as they go in the document).
DOMContentLoaded	Scripts with async may load and execute while the document has not yet been fully downloaded.	Scripts with defer execute after the document is loaded and parsed (they wait if needed), right before DOMContentLoaded.

[DAN.IT] EDUCATION

[window onload]

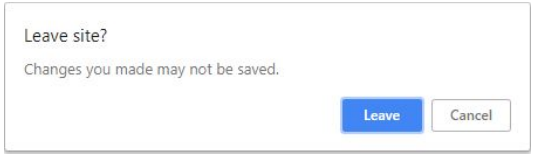
- The **load** event on the window object triggers when the whole page is loaded including styles, images and other resources
- The example below correctly shows image sizes, because window.onload waits for all images:

```
<script>
    window.onload = function ()
        {  alert("Page loaded");

        // image is loaded at this time
        alert(`Image size: ${img.offsetWidth}x${img.offsetHeight}`);
    };
</script>

<img id="img" src="https://en.js.cx/clipart/train.gif?speed=1&cache=0">
```

[window unload]

- When a visitor leaves the page, the **unload** event triggers on window
  - We can do there actions that don't involve a delay, like closing popup windows
  - But we can't cancel the transition to another page
- For that we should use another event – **onbeforeunload**
  - In this event you can ask the user for additional confirmation before leaving the page
  - Some browsers disable this feature, because certain webmasters abused this event handler by showing misleading and hackish messages
- Try it by running the following code:

```
<script>
    window.onbeforeunload = function () {
        return "There are unsaved changes. Leave now?";
    };
</script>

<a href="http://example.com">Leave for EXAMPLE.COM</a>
```



[DAN.IT] EDUCATION

[Resource Loading Events]

- The browser allows to track the loading of external resources – scripts, images, iframes and so on
- There are two events for it:
  - **onload** – successful load
  - **onerror** – an error occurred

[DAN.IT] EDUCATION

[Loading a Script]

- Let's say we need to call a function that resides in an external script
- We can load it dynamically, like this:

```
// Load the jquery script
let script = document.createElement("script");
script.src = "https://code.jquery.com/jquery-3.3.1.js";
document.head.append(script);

alert($);  // Uncaught reference error
```

- We need to wait until the script loads, and only then we can call it
- The main helper is the **load** event - it triggers after the script was loaded and executed

[Loading a Script]

- In **onload** we can use script variables, run functions etc:

```
let script = document.createElement("script");
script.src = "https://code.jquery.com/jquery-3.3.1.js";
document.head.append(script);

script.onload =  function () {
    // The script creates a helper function "$"
    alert($);
}
```

- Errors that occur during the loading of the script can be tracked on **error** event
- For instance, let's request a script that doesn't exist:

```
script = document.createElement("script");
script.src = "https://example.com/404.js"; // no such script
document.head.append(script);

script.onerror = function () {
    alert("Error loading " + this.src); // Error loading
    https://example.com/404.js
};
```

[DAN.IT] EDUCATION

[Exercise (7)]

- Normally, images are loaded when they are created
- So when we add <img> to the page, the user does not see the picture immediately
- To show an image immediately, we can create it “in advance”, like this:

```
let img = new Image();
img.src = 'my.jpg';
```

  - The browser starts loading the image and remembers it in the cache
  - Later, when the same image appears in the document, it shows up immediately
- Create a function **preloadImages(sources, callback)** that loads all images from the array sources and, when ready, runs callback
- For instance, this will show an alert after the images are loaded:

```
function loaded() {
    alert("Images loaded")
}
preloadImages(["1.jpg", "2.jpg", "3.jpg"], loaded);
```

[DAN.IT] EDUCATION

[Exercise (7)]

→ Use the following code to test your function:

```
function preloadImages(sources, callback) {
    /* your code */
}

// ---------- The test ----------
let sources = [
    "https://en.js.cx/images-load/1.jpg,
    "https://en.js.cx/images-load/2.jpg,
    "https://en.js.cx/images-load/3.jpg"
];

// add random characters to prevent browser
caching  for (let i = 0; i < sources.length; i++){
    sources[i] += '?' + Math.random();
}
```

```
// for each image, let's create another img
with  the same src and check that we have its
width  immediately
function testLoaded() {
    let widthSum = 0;
    for (let i = 0; i < sources.length; i++) {
        let img =
        document.createElement('img');
        img.src = sources[i];
        widthSum += img.width;
    }
    alert(widthSum);
}

// every image is 100x100, the total width
should  be 300
preloadImages(sources, testLoaded);
```

[DAN.IT] EDUCATION

[Form Properties and Methods]

- Forms and control elements, such as <input> have a lot of special properties and events
- Working with forms can be much more convenient if we know them
- Document forms are members of the special collection **document.forms**
- That's a **named collection**: we can use both the name and the number to get the form:
  - document.forms.my – the form with name "my"
  - document.forms[0] – the first form in the document
- Any element in a form is available in the named collection **form.elements**

[DAN.IT] EDUCATION

[Form Properties and Methods]

- For example:

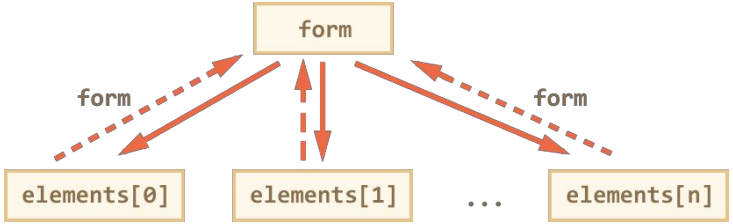
```
<form name="myform">
    <input  name="one" value="1">
    <input  name="two" value="2">
</form>

<script>
    // get the form
    let form = document.forms.myform;

    // get the element
    let elem = form.elements.one;

    alert(elem.value); // 1
</script>
```

- There's a shorter notation: we can access the element as form[index/name]
  - e.g., instead of form.elements.one we can write form.one

[Backreference: element.form]

- For any element, the form is available as **element.form**
- So a form references all elements, and elements reference theform
- For example:

```
<form id="form1">
    <input type="text" name="login">
</form>

<script>
    // form -> element
    let login = form1.login;

    // element -> form
    alert(login.form); //
    HTMLFormElement
</script>
```



[Input and Textarea]

→ Normally, we can access the value as input.value or input.checked for checkboxes

```
<form>
    <input type="text" id="txt1"><br />
    <input type="checkbox" id="chk1" /><br />
    <textarea id="area1"></textarea><br />
</form>

<script>
    txt1.value = "New value";
    chk1.checked = true;
    area1.value = "New text";
</script>
```

New value

New text

[Select and Options]

- A <select> element has 3 important properties:
  - **select.options** – the collection of <option> elements
  - **select.value** – the value of the chosen option
  - **select.selectedIndex** – the number of the selected option
- So we have three ways to set the value of a <select>:
  1. Find the needed <option> and set option.selected to true
  2. Set select.value to the value
  3. Set select.selectedIndex to the number of the option
  - The first way is the most obvious, but (2) and (3) are usually more convenient

[DAN.IT] EDUCATION

[Select and Options]

→ Here is an example:

```
<select id="select">
    <option value="apple">Apple</option>
    <option value="pear">Pear</option>
    <option value="banana">Banana</option>
</select>

<script>
    // all three lines do the same thing
    select.options[2].selected = true;
    select.selectedIndex = 2;
    select.value = "banana";
</script>
```

Banana ▾

[Select with Multiple Choice]

- Unlike most other controls, **<select multiple>** allows multiple choice
- In that case we need to walk over select.options to get all selected values:

```
<select id="genres" multiple>
    <option value="blues" selected>Blues</option>
    <option value="rock" selected>Rock</option>
    <option value="classic">Classic</option>
</select>

<script>
    // get all selected values from multi-select
    let selected = Array.from(genres.options)
        .filter(option => option.selected)
        .map(option => option.value);

    alert(selected); // blues,rock
</script>
```

[DAN.IT] EDUCATION

[Creating a New Option]

- In the specification of the option element, there's a nice short syntax to create <option> elements:

```
option = new Option(text, value, defaultSelected, selected);
```

  - text – the text inside the option
  - value – the option value
  - defaultSelected – if true, then selected attribute is created
  - selected – if true, then the option is selected
- Call the select.options.add() method to add the new option
- For instance:

```
let option = new Option("Text", "value", true, true);
// creates <option value="value" selected>Text</option>

select.options.add(option);
```

[DAN.IT] EDUCATION

[Option Properties]

- Option elements have additional properties:
  - **selected** – is the option selected
  - **index** – the number of the option among the others in its <select>
  - **text** – the text content of the option (seen by what the visitor)

[Exercise (8)]

- There's a <select>:

```
let option = new Option("Text", "value");
// creates <option value="value">Text</option>

let option = new Option("Text", "value", true, true);
// creates <option value="value" selected>Text</option>
```

- Use JavaScript to:
  - Show the value and the text of the selected option
  - Add an option: <option value="classic">Classic</option>
  - Make it selected

[Focus Events]

- The **focus** event is called when an element receives a focus
  - This occurs when the user either clicks on it or uses the Tab key on the keyboard
- Focusing generally means: “prepare to accept the data here”, so that’s when we can run code to initialize or load something
- The **blur** event is called when an element loses the focus
  - This occurs when the user clicks somewhere else or presses Tab to go to the next form field
- Losing the focus generally means: “the data has been entered”, so we can run code to check it or even to save it to the server and so on

[DAN.IT] EDUCATION

[Focus Events]

- Let's use the focus events for validation of an input field
- In the example below:
  - The blur handler checks if the field the email is entered, and if not – shows an error.
  - The focus handler hides the error message (on blur it will be checked again)

```
<style>
    .invalid {
        border-color: red;
    }
    #error {
        color: red;
    }
</style>

Your email please: <input type="email" id="input">
<span id="error" hidden>Please enter a valid email</span><br/>
<button>Send</button>
```

[Focus Events]

```
<script>
    input.onblur = function () {
        if (!input.value.includes('@')) { // not email
            input.classList.add("invalid");
            error.hidden = false;
        }
    };

    input.onfocus = function () {
        if (this.classList.contains("invalid")) {
            // remove the "error" indication, because
the user wants to re-enter something
            this.classList.remove("invalid");
            error.hidden = true;
        }
    };
 </script>
```

Your email please: a Please enter a valid email

Send

[Methods focus/blur]

- Methods **elem.focus()** and **elem.blur()** set/unset the focus on the element
- For instance, let's make the visitor unable to leave the input if the value is invalid:

```
input.onblur = function () {
    if (!input.value.includes('@')) { // not email
        input.classList.add("invalid");
        error.hidden = false;

        // put the focus back
        input.focus();
    }
    else {
        this.classList.remove("invalid");
        error.hidden = true;
    }
};
```

- Note that we can't "prevent losing focus" by calling event.preventDefault() in onblur, because onblur works *after* the element lost the focus

[Allow Focusing on Any Element]

- Many elements do not support focusing by default
- focus/blur support is guaranteed for elements that a visitor can interact with
  - <button>, <input>, <select>, <a>, etc.
- On the other hand, elements that exist to format something such as , , <table> are unfocusable by default
  - The method elem.focus() doesn't work on them, and focus/blur events are never triggered
- This can be changed using HTML-attribute **tabindex**
  - Any element supports focusing if it has tabindex
- tabindex specifies the order number of the element when Tab is used to move between elements
  - If we have two elements, the first has tabindex="1", and the second has tabindex="2", then pressing Tab while in the first element – moves us to the second one

[DAN.IT] EDUCATION

[Allow Focusing on Any Element]

- tabindex has two special values:
  - tabindex="0" makes the element the last one
  - tabindex="-1" means that Tab should ignore that element

```
<style>
    li {
        cursor: pointer;
    }
    :focus {
        outline: 1px dashed green;
    }
</style>
Click the first item and press Tab. Keep track of the
order.
<ul>
    <li tabindex="1">One</li>
    <li tabindex="0">Zero</li>
    <li tabindex="2">Two</li>
    <li tabindex="-1">Minus one</li>
</ul>
```

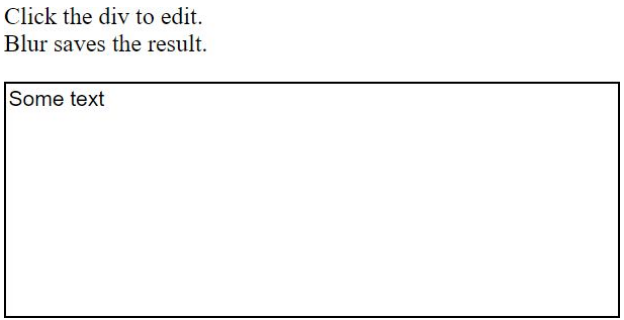
Click the first item and press Tab. Keep track of the order.

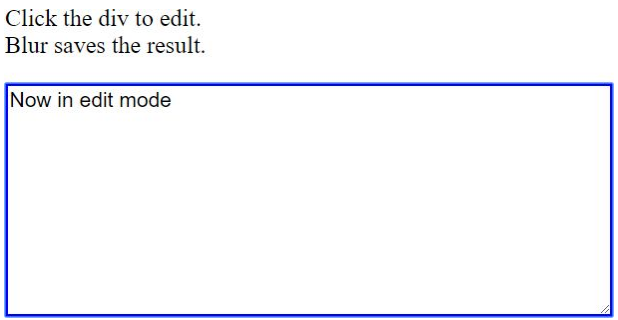
- One
- Zero
- Two
- Minus one

- Normally, <li> does not support focusing, but tabindex full enables it, along with events and styling

[Exercise (9)]

- Create a that turns into <textarea> when clicked
- The textarea allows to edit the text in the
- When the textarea looses focus, it turns back into , and its content becomes the HTML in s





- Start with the code on the following slide

[Exercise (9)]

```
<style>
    /* Make the div and the textarea the same size */
    .view, .edit {
        height: 150px;
        width: 400px;
        font-family: arial;
        font-size: 14px;
    }
    .view {
        border: 1px solid black;
        padding: 2px;
    }
    .edit {
        display: block;
        border: 2px solid blue;
        padding: 1px;
    }
    ul {
        padding: 0;
    }
 </style>
```

```
<ul>
    <li>Click the  div to edit.</li>
    <li>Blur  saves  the result.</li>
</ul>

<div id="view" class="view">Some text</div>

<script>
    // ...your code...
    // Note: <textarea> should have class="edit"
</script>
```

[Change Event]

- The **change** event triggers when the element has finished changing
- For text inputs that means that the event occurs when it looses focus
- For other elements: select, input type=checkbox/radio it triggers right after the selection changes
- In the following example when we move the focus from the text field, for instance, click on a button – there will be a change event:

```
<input type="text" onchange="alert(this.value)">
<input type="button" value="Button">
```

Button

[Input Event]

- The **input** event triggers every time a value is modified
- Unlike keyboard events it works on any value change, even those that do not involve keyboard actions: pasting with a mouse or using speech recognition
- The input event occurs after the value is modified, so we can't use event.preventDefault() there – it's just too late, there would be no effect

```
<input type="text" onchange="alert(this.value)">
<input type="button" value="Button">
```

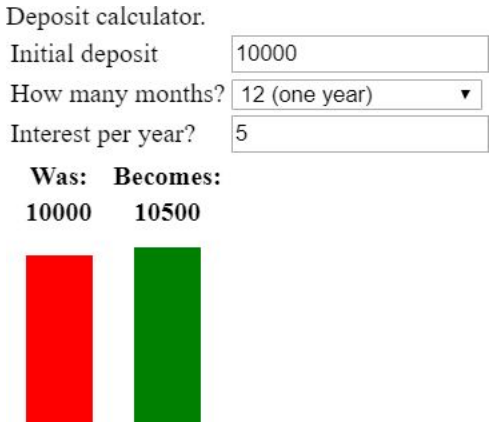
test oninput: test

[Exercise (10)]

- Create an interface that allows to enter a sum of bank deposit and percentage, then calculates how much it will be after given periods of time
- Any input change should be processed immediately
- The formula is:

```
// initial: the initial money sum
// interest: e.g. 0.05 means 5% per year
// years: how many years to wait
let result = Math.round(initial * (1 + interest * years));
```

- Start with the HTML on the next slides



[Exercise (10)]

```html
<!DOCTYPE html>
<html>

<head>
    <meta charset="utf-8">
    <style>
        td select,
        td input {
            width: 150px;
        }

        #diagram td {
            vertical-align: bottom;
            text-align: center;
            padding: 10px;
        }

        #diagram div {
            margin: auto;
        }
    </style>
</head>
```

[Exercise (10)]

```
<body>
        Deposit calculator.
    <form name="calculator">
        <table>
                <tr>
                <td>Initial deposit</td>
                <td>
                    <input name="money" type="number" value="10000" required>
                </td>
            </tr>
                <tr>
                <td>How many months?</td>
                <td>
                    <select name="months">
                        <option value="3">3 (minimum)</option>
                        <option value="6">6 (half-year)</option>
                        <option value="12" selected>12 (one year)</option>
                        <option value="18">18 (1.5 years)</option>
                        <option value="24">24 (2 years)</option>
                        <option value="30">30 (2.5 years)</option>
                        <option value="36">36 (3 years)</option>
                         <option value="60">60 (5 years)</option>
                     </select>
                 </td>
             </tr>
             <tr>
                <td>Interest per year?</td>
                <td>
                   <input name="interest" type="number" value="5"
                   required>
                </td>
            </tr>
        </table>
    </form>
```

[Exercise (10)]

```
<table id="diagram">
        <tr>
            <th>Was:</th>
            <th>Becomes:</th>
        </tr>
        <tr>
            <th id="money-before"></th>
            <th id="money-after"></th>
        </tr>
        <tr>
            <td>
                <div style="background: red;width:40px;height:100px"></div>
            </td>
            <td>
                <div style="background: green;width:40px;height:0" id="height-after"></div>
            </td>
        </tr>
    </table>

    <script>
        let form = document.forms.calculator;
        // your code
    </script>
</body>
</html>
```

[DAN.IT] EDUCATION

[Form Submission]

- There are two main ways to submit a form:
  - The first – to click <input type="submit"> or <input type="image">
  - The second – press Enter on an input field
- The **submit** event triggers when the form is submitted
- It is usually used to validate the form before sending it to the server
- The submit handler can check the data, and if there are errors, show them and call event.preventDefault(), then the form won't be sent to the server

[DAN.IT] EDUCATION

[Form Submission]

→ Here is an example:

```
<form id="form" method="get">
    Enter your name: <input type="text" id="name" name="name"/><br />
    Enter your age: <input type="number" id="age" name="age" /><br />
    <input type="submit" value="Submit">
</form>

<script>
    form.onsubmit = function (event) {
        let name = document.getElementById("name").value;
        if (!name) {
            alert("You must enter your first name");
            event.preventDefault();
            return;
        }
        let age = Number(document.getElementById("age").value);
        if (age < 0 || age > 120) {
            alert("Age must be between 0 and 120");
            event.preventDefault();
            return;
        }
    }
</script>
```

Enter your name:

Enter your age:

Submit

[DAN.IT] EDUCATION

[Form Submission]

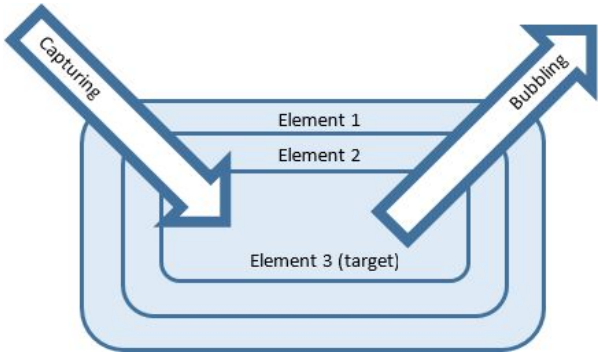
- To submit a form to the server manually, we can call **form.submit()**
- Then the submit event is not generated
  - It is assumed that if the programmer calls form.submit(), then the script already did all related processing
- Sometimes that's used to manually create and send a form, like this:

```
googleSearch.onclick = function () {
    let form = document.createElement("form");
    form.action = "https://google.com/search";
    form.method = "GET";

    form.innerHTML = '<input name="q" value="test">';

    // the form must be in the document to submit it
    document.body.append(form);

    form.submit();
};
```

[DAN.IT] EDUCATION

[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

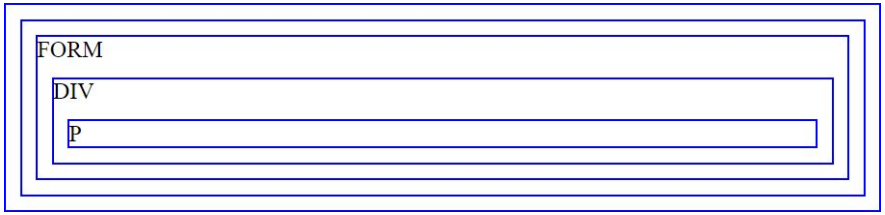


[DAN.IT]
EDUCATION

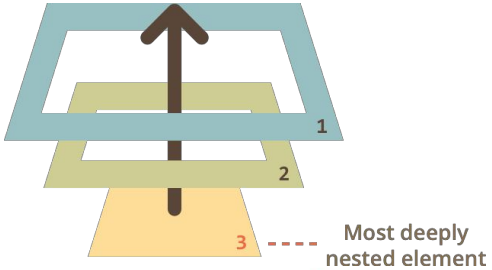
[Bubbling]

- 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
  - Then on the outer <form>
  - And so on upwards till the document object



[event.target]

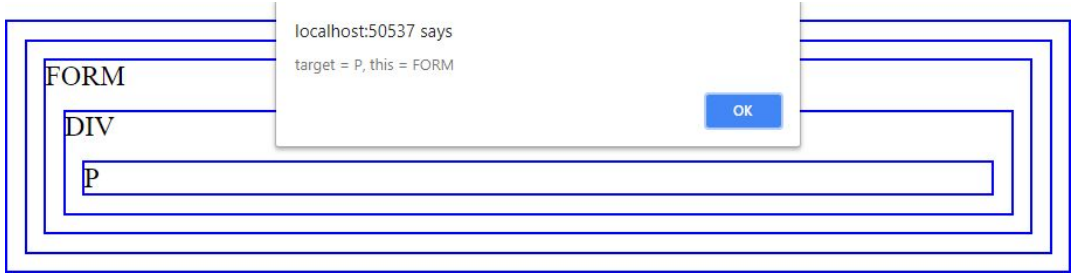
- A handler on a parent element is able to know where it actually happened
- The most deeply nested element that caused the event is called a *target* element, accessible as **event.target**
- Note the differences from **this** (=event.currentTarget):
  - event.target – is the “target” element that initiated 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

[DAN.IT] EDUCATION

[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:



[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>:

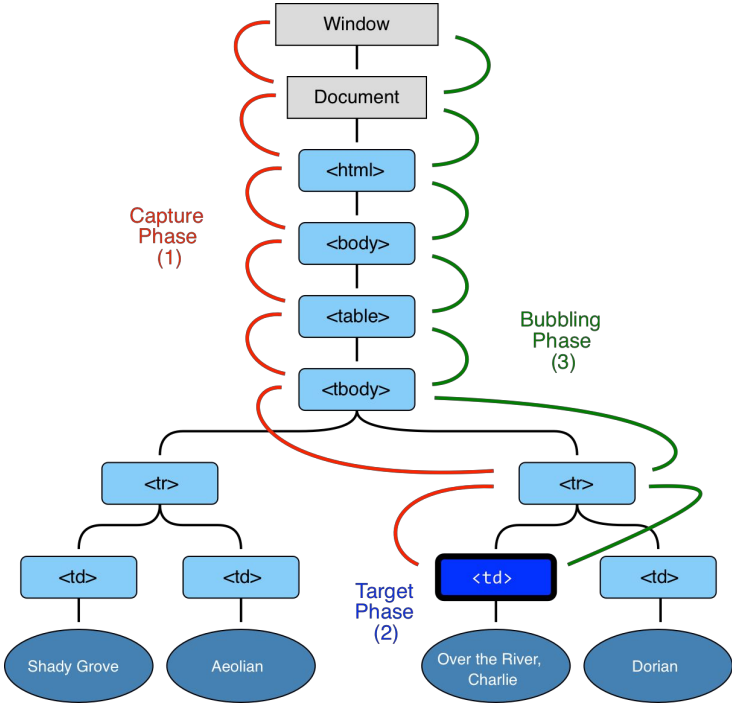
```
<body onclick="alert(`the bubbling doesn't reach here`)">
    <button onclick="event.stopPropagation()">Click me</button>
</body>
```

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

[DAN.IT] EDUCATION

[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



[DAN.IT] EDUCATION

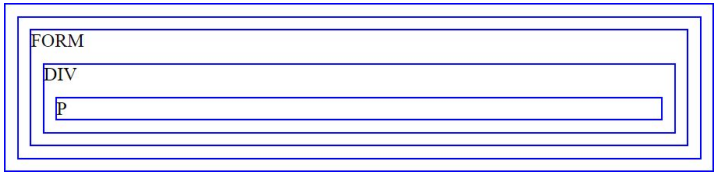
[Capturing]

- 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

[DAN.IT] EDUCATION

[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)

[DAN.IT]
EDUCATION

[Event Delegation]

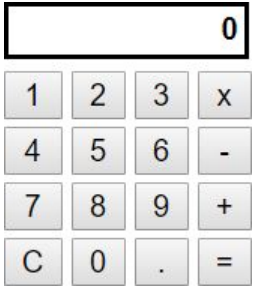
- Capturing and bubbling allow us to implement one of most powerful event handling patterns called **event delegation**
- The idea is that if we have a lot of elements handled in a similar way, then instead of assigning a handler to each of them – we put a single handler on their common ancestor
- In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

[DAN.IT] EDUCATION

[Event Delegation]

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

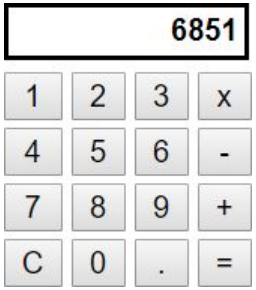
```
<div id="calculator">
    <input id="ans" type="text" value="0" />
    <table id="table">
        <tr>
            <td><button>1</button></td>
            <td><button>2</button></td>
            <td><button>3</button></td>
            <td><button>x</button></td>
        </tr>
        <tr> … </tr>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

[DAN.IT] EDUCATION

[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.

[DAN.IT] EDUCATION

[Exercise (11)]

→ 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

[DAN.IT] EDUCATION

[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:

```
<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>
```

HTML CSS JavaScript

- 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 , 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.

[DAN.IT] EDUCATION

[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 (12)]

- 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 ....
- Start with the code on the following slide

[Exercise (12)]

```
<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>
```

[Control questions]

1. What is Event?
2. How can we attach an event on the page?
3. How can we detect which key was pressed?
4. What event is triggered when page DOM is ready?
5. How can we set a select-element value?
6. What event react to input-element changes?
7. How can we process form-element on its submit?
8. What is bubbling?
9. How can we stop event propagation?
10. When can we use event delegation?

[DAN.IT] EDUCATION