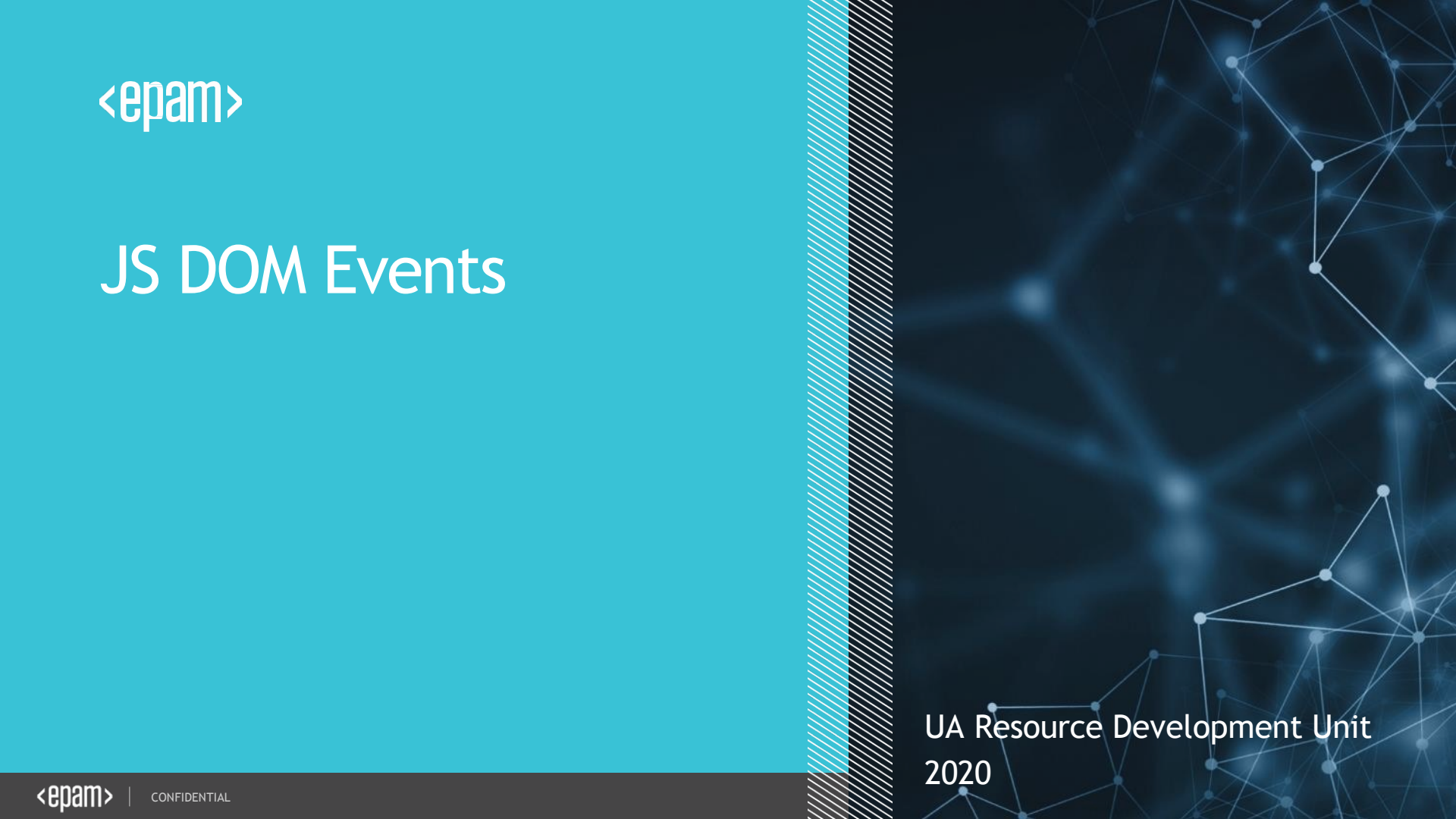

JS DOM Events
UA Resource Development Unit
2020

AGENDA

1. Definition
2. Types of Events
3. Event Handlers
4. Event Phase
5. Custom Events

event

An event is a signal that something has happened.

Types of Events

- form events,
- mouse events,
- key events,
- HTML5 Events,
- Touchscreen and Mobile Events

Mouse events

- 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.

Keyboard Events

onkeydown

The event occurs when the user is pressing a key

onkeypress

The event occurs when the user presses a key

onkeyup

The event occurs when the user releases a key

oninput

The DOM input event fires synchronously when the value of an <input>, <select>, or <textarea> element has been altered.

onfocus

The focus event fires when an element has received focus.

onblur

The blur event fires when an element has lost focus.

Other

- Form elements like on submit, focus or change
- Media events like onplay, on pause, on onseeked
- Drag events like ondrag, ondragenter, ondrop
- General object events like onload, onresize, onscroll
- Other – clipboard, print, touch, transition, speech recognition

Form element events

submit – when the visitor submits a <form>.

reset – The reset event fires when a <form> is reset.

formdata – The formdata event fires after the entry list representing the form's data is constructed.

More details: https://developer.mozilla.org/en-US/docs/Web/API/HTMLFormElement#events

Event handlers

- HTML-attribute
- DOM property
- Accessing the element: this
- addEventListener

Event handlers

```
<!-- Inline -->
<div onclick="getACupcake(event)"><div>
addEventListener()


// addEventListener()
document.getElementById('cupcakeButton').addEventListener('click', getACupcake);

// Assigning to event handler properties
document.getElementById('cupcakeButton').onclick = getACupcake;
```

HTML-attribute

A handler can be set in HTML with an attribute named **on<event>**.

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

DOM property

We can assign a handler using a DOM property on<event>

```
<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.

As there's only one onclick property, we can't assign more than one event handler.

Accessing the element: this

The value of this inside a handler is the element. The one which has the handler on it.

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

addEventListener

```
input.onclick = function() { alert(1); }
// ...
input.onclick = function() { alert(2); }
```

we can’t assign multiple handlers to one event.

addEventListener

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

event

Event name, e.g. "click".

handler

The handler function.

phase

An optional argument, the “phase” for the handler to work

removeEventListener

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

```
function handler() {
  alert( 'Thanks!' );
}

input.addEventListener("click", handler);
// ....
input.removeEventListener("click", handler);
```

If we don’t store the function in a variable, then we can’t remove it. There’s no way to “read back” handlers assigned by addEventListener

Event object

event.type

Event type, above is "click".

event.currentTarget

Element that handled the event. That’s exactly the same as this, unless you bind this to something else, and then event.currentTarget becomes useful.

event.clientX / event.clientY

Window-relative coordinates of the cursor, for mouse events.

Event object

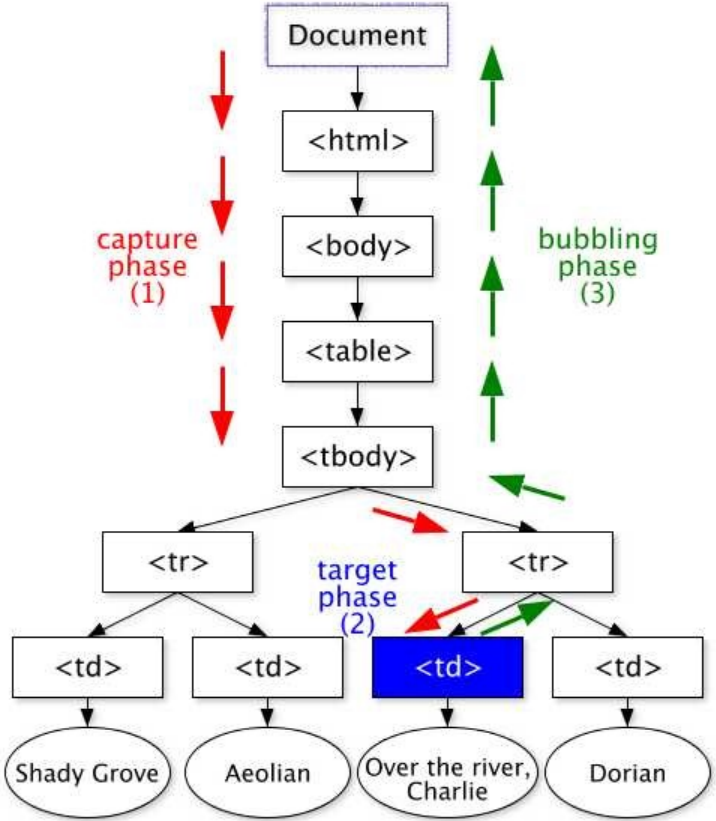
event.target is the original element the event happened to.

```
function getACupcake(event) {
  event.target.style.backgroundColor = '#F00'; // use it
}
```

The event object is also accessible from HTML

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

Bubbling & Capturing



1. Capturing phase – the event goes down to the element.
2. Target phase – the event reached the target element.
3. Bubbling phase – the event bubbles up from the element.

Bubbling

When an event happens on an element, it first runs the handlers on it, then on its parent, then all the way up on other ancestors.

```
document.addEventListener("DOMContentLoaded", function () {
  let allElements = document.getElementsByTagName("*");
  for (var i = 0; i < allElements.length; i++) {
    allElements[i].addEventListener("click",
      function () { console.log(this.tagName); }, false);
  };
  document.addEventListener("click",
    function () { console.log(this); }, false);
  window.addEventListener("click",
    function () { console.log(this); }, false);
});
```

A bubbling event goes from the target element straight up. Normally it goes upwards till <html>, and then to documentobject, and some events even reach window, calling all handlers on the path.

Stopping bubbling

Any handler may decide that the event has been fully processed and stop the bubbling.

```
// Prevents further propagation of the current
// event in the capturing and bubbling phases.
event.stopPropagation()
// Prevents other listeners of the same event from being called
event.stopImmediatePropagation()
```

If several listeners are attached to the same element for the same event type, they are called in order in which they have been added. If during one such call, **event.stopImmediatePropagation()** is called, no remaining listeners will be called.

event.stopPropagation()
event.stopImmediatePropagation()

In other words, event.stopPropagation() stops the move upwards, but on the current element all other handlers will run.

To stop the bubbling and prevent handlers on the current element from running, there's a method event.stopImmediatePropagation(). After it no other handlers execute.

```
<body>
    <div class="overlay">
        Click outside to close.
    </div>
</body>
```

```
var overlay = document.querySelector('.overlay');

overlay.addEventListener('click', function (event) {
  event.stopPropagation();
});
// Remove the overlay when a 'click'
// is heard on the document (<html>) element
document.addEventListener('click', function (event) {
  overlay.parentNode.removeChild(overlay);
});
```

Capturing

Handlers added using on<event>-property or using HTML attributes or using **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.

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.

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, bubbling=3).

Event delegation

The algorithm:

Put a single handler on the container.

In the handler – check the source element event.target.

If the event happened inside an element that interests us, then handle the event.

```
<ul id="parent-list">
    <li id="post-1">Item 1</li>
    <li id="post-2">Item 2</li>
    <li id="post-3">Item 3</li>
    <li id="post-4">Item 4</li>
    <li id="post-5">Item 5</li>
    <li id="post-6">Item 6</li>
</ul>
```

Event delegation

Benefits:

Simplifies initialization and saves memory: no need to add many handlers.

Less code: when adding or removing elements, no need to add/remove handlers.

DOM modifications: we can mass add/remove elements with innerHTML and alike.

```
document.getElementById("parent-list").addEventListener("click", function (e) {
  // e.target is the clicked element!
  // If it was a list item
  if (e.target && e.target.nodeName == "LI") {
    // List item found!  Output the ID!
    console.log("List item ", e.target.id.replace("post-", ""), " was clicked!");
  }
});
```

Browser default actions

All the default actions can be prevented if we want to handle the event exclusively by JavaScript.
To prevent a default action – use either event.preventDefault() or return false. The second method works only for handlers assigned with on<event>.
If the default action was prevented, the value of event.defaultPrevented becomes true, otherwise it's false.

```
function stopDefAction(ev) {
  ev.preventDefault();
}

document.getElementById('my-checkbox').addEventListener(
  'click', stopDefAction, false
);
```

Dispatching custom events

```
let event = new Event(event type[, options]);
```

Arguments:

event type – may be any string, like "click" or our own like "hey-ho!".

options – the object with two optional properties:

- bubbles: true/false – if true, then the event bubbles.
- cancelable: true/false – if true, then the “default action” may be prevented. Later we’ll see what it means for custom events.

By default both are false: {bubbles: false, cancelable: false}

dispatchEvent()

Dispatches an Event at the specified EventTarget, invoking the affected EventListeners in the appropriate order. The normal event processing rules (including the capturing and optional bubbling phase) also apply to events dispatched manually with dispatchEvent().

```
cancelled = !target.dispatchEvent(event)
```

- event is the Event object to be dispatched.
- target is used to initialize the Event.target and determine which event listeners to invoke.

- The return value is false if event is cancelable and at least one of the event handlers which handled this event called Eve

Creating custom events

```
var event = new Event('build');

// Listen for the event.
elem.addEventListener('build', function (e) { ... }, false);

// Dispatch the event.
elem.dispatchEvent(event);
```

Adding custom data – CustomEvent()

To add more data to the event object, the CustomEvent interface exists and the **detail** property can be used to pass custom data

```
var event = new CustomEvent('build', { detail: elem.dataset.time });

function eventHandler(e) {
  console.log('The time is: ' + e.detail);
}
```

Useful Link

An Introduction to the Document Object Model (DOM) 🚀

https://javascriptforwp.com/the-dom/

https://javascriptforwp.com/creating-and-adding-text-and-element-nodes-to-the-dom/

https://javascriptforwp.com/intro-to-events/

https://javascriptforwp.com/dom-event-propagation-javascript-capturing-bubbling-explained/

JavaScript Event Capture, Propagation and Bubbling 🔥

https://youtu.be/F1anRyL37lE

Q&A

UA Frontend Online LAB