What is DOM Event delegation?

Asked 15 years, 1 month ago Modified 1 year, 11 months ago

Viewed 154k times

274

Can anyone please explain event delegation in JavaScript and how is it useful?

javascript event-handling dom-events event-delegation

Share

Improve this question

Follow

edited Aug 26, 2019 at 13:27

Brian Tompsett - 汤莱恩

5,875 ● 72 ● 61 ● 133

asked Nov 6, 2009 at 12:33

Xenon

2,757 ● 3 ● 16 ● 4

3 It's be nice if there was a link to smoe useful source of information about this. 6 hours in, this is the google top hit for "dom event delegation". Maybe this is a useful link? I'm not entirely sure: w3.org/TR/DOM-Level-2-Events/events.html – Sean McMillan Nov 6, 2009 at 19:05

Or maybe this: sitepoint.com/blogs/2008/07/23/… – Sean McMillan Nov 6, 2009 at 19:07

9 This is a popular one. Even fb guys link to this for their reactjs page davidwalsh.name/event-delegate – Sorter Nov 9, 2015 at 10:24

See this javascript.info/event-delegation it will help you a lot – Suraj Jain Apr 9, 2018 at 2:40

10 Answers

Sorted by: Highest score (default)

415

DOM event delegation is a mechanism of responding to ui-events via a single common parent rather than each child, through the magic of event "bubbling" (aka event propagation).

When an event is triggered on an element, the following occurs:

> The event is dispatched to its target `EventTarget` and any event listeners found there are triggered. *Bubbling* events will then trigger any additional event listeners found by following the `EventTarget`'s parent chain *upward*, checking for any event listeners registered on each successive EventTarget. This upward propagation will continue up to and including the `Document`.

Event bubbling provides the foundation for event delegation in browsers. Now you can bind an event handler to a single parent element, and that handler will get executed whenever the event occurs *on any of its child nodes* (and any of their children in turn). **This is event delegation.** Here's an example of it in practice:

```
<ul onclick="alert(event.type + '!')">
    <li>One</li>
    <li>Two</li>
    <li>Three</li>
</ul>
```

With that example if you were to click on any of the child `<li>` nodes, you would see an alert of `"click!"`, even though there is no click handler bound to the `<li>` you clicked on. If we bound `onclick="..."` to each `<li>` you would get the same effect.

So what's the benefit?

Imagine you now have a need to dynamically add new `<li>` items to the above list via DOM manipulation:

```
var newLi = document.createElement('li');
newLi.innerHTML = 'Four';
myUL.appendChild(newLi);
```

Without using event delegation you would have to "rebind" the `"onclick"` event handler to the new `<li>` element, in order for it to act the same way as its siblings. *With* event delegation you don't need to do anything. Just add the new `<li>` to the list and you're done.

This is absolutely fantastic for web apps with event handlers bound to many elements, where new elements are dynamically created and/or removed in the DOM. With event delegation the number of event bindings can be drastically decreased by moving them to a common

parent element, and code that dynamically creates new elements on the fly can be decoupled from the logic of binding their event handlers.

Another benefit to event delegation is that the total memory footprint used by event listeners goes down (since the number of event bindings go down). It may not make much of a difference to small pages that unload often (i.e. user's navigate to different pages often). But for long-lived applications it can be significant. There are some really difficult-to-track-down situations when elements removed from the DOM still claim memory (i.e. they leak), and often this leaked memory is tied to an event binding. With event delegation you're free to destroy child elements without risk of forgetting to "unbind" their event listeners (since the listener is on the ancestor). These types of memory leaks can then be contained (if not eliminated, which is freaking hard to do sometimes. IE I'm looking at you).

Here are some better concrete code examples of event delegation:

- How JavaScript Event Delegation Works
- Event Delegation versus Event Handling
- jQuery.delegate is event delegation + selector specification
- jQuery.on uses event delegation when passed a selector as the 2nd parameter
- Event delegation without a JavaScript library

- Closures vs Event delegation: takes a look at the pros of *not* converting code to use event delegation
- Interesting approach PPK uncovered for delegating the `focus` and `blur` events (which do *not* bubble)

Share Improve this answer

Follow

edited Jan 13, 2023 at 8:38

dumbass

27.2k ● 4 ● 35 ● 72

answered Nov 6, 2009 at 15:23

Crescent Fresh

117k ● 27 ● 157 ● 140

I have got access forbidden on opening your third link Event delegation without a javascript library and +1 for your last link – bugwheels94 Jul 15, 2013 at 12:46

Hello, thank you for a great explanation. I am still confused about a certain detail though: The way I understand the DOM tree event flow (As can be seen in 3.1. Event dispatch and DOM event flow) the event object propagates until it reaches the target element then bubbles up. How come it can reach child elements of a node if the parent of this node is the event target in question? e.g. how can the event propagate to a `<li>` when it should stop at `<ul>`? If my question is still unclear or needs a separate thread I'd be happy to oblige. – Imad Apr 11, 2017 at 9:14

@Aetos: > *How come it can reach child elements of a node if the parent of this node is the event target in question?* It cannot, as I understand it. The event finishes phase 1 (capturing) at the parent of the target, enters phase 2 (target) on the target itself, then enters phase 3 (bubbling) starting on the parent of the target. Nowhere does it reach a child of the target. – Crescent Fresh Apr 12, 2017 at 15:01

@Crescent Fresh well then how does the event apply on the child node if it never reaches it? – Imad Apr 12, 2017 at 15:19

@Aetos: I see where you're coming from. I think the confusion stems from thinking that it is the `onclick=...` that sets up event propagation. It's not. The spec is defining what a DOM/browser implementation *should do* when an event occurs. A "click" event (for example) is dispatched by the implementation as a result of a user action, irrespective of `onclick=...` being present in the code or not! It is only during the propagation of this event that the implementation should look for event listeners and invoke them. The `onclick=` simply registers a listener... – Crescent Fresh Apr 12, 2017 at 17:41

52

Event delegation allows you to avoid adding event listeners to specific nodes; instead, the event listener is added to one parent. That event listener analyzes bubbled events to find a match on child elements.

JavaScript Example :

Let's say that we have a parent UL element with several child elements:

```
<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>
```

Let's also say that something needs to happen when each child element is clicked. You could add a separate event listener to each individual LI element, but what if LI elements are frequently added and removed from the list? Adding and removing event listeners would be a nightmare, especially if addition and removal code is in different places within your app. The better solution is to add an event listener to the parent UL element. But if you add the event listener to the parent, how will you know which element was clicked?

Simple: when the event bubbles up to the UL element, you check the event object's target property to gain a reference to the actual clicked node. Here's a very basic JavaScript snippet which illustrates event delegation:

```
// Get the element, add a click listener...
document.getElementById("parent-list").addEventListene
    // 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(
clicked!");
    }
});
```

Start by adding a click event listener to the parent element. When the event listener is triggered, check the event element to ensure it's the type of element to react to. If it is an LI element, boom: we have what we need! If it's not an element that we want, the event can be ignored. This example is pretty simple -- UL and LI is a

straight-forward comparison. Let's try something more difficult. Let's have a parent DIV with many children but all we care about is an A tag with the classA CSS class:

```
// Get the parent DIV, add click listener...
document.getElementById("myDiv").addEventListener("cli
    // e.target was the clicked element
    if(e.target && e.target.nodeName == "A") {
        // Get the CSS classes
        var classes = e.target.className.split(" ");
        // Search for the CSS class!
        if(classes) {
            // For every CSS class the element has...
            for(var x = 0; x < classes.length; x++) {
                // If it has the CSS class we want...
                if(classes[x] == "classA") {
                    // Bingo!
                    console.log("Anchor element clicke
                    // Now do something here....
                }
            }
        }
    }
});
```

http://davidwalsh.name/event-delegate

Share Improve this answer

Follow

edited Jan 13, 2023 at 8:38

dumbass

27.2k ● 4 ● 35 ● 72

answered Oct 8, 2015 at 12:31

Osama AbuSitta

4,066 ● 4 ● 37 ● 54

5 Suggested tweak: use e.classList.contains() instead in the last example: developer.mozilla.org/en-

US/docs/Web/API/Element/classList – nc. Sep 18, 2016 at 6:33

9

dom event delegation is something different from the computer science definition.

It refers to handling bubbling events from many elements, like table cells, from a parent object, like the table. It can keep the code simpler, especially when adding or removing elements, and saves some memory.

Share Improve this answer

Follow

answered Nov 6, 2009 at 14:24

kennebec

105k ● 32 ● 108 ● 127

9

To understand event delegation first we need to know why and when we actually need or want event delegation.

There may be many cases but let's discuss two big use cases for event delegation. 1. The first case is when we have an element with lots of child elements that we are interested in. In this case, instead of adding an event handler to all of these child elements, we simply add it to the parent element and then determine on which child element the event was fired.

2.The second use case for event delegation is when we want an event handler attached to an element that is not yet in the DOM when our page is loaded. That's, of course, because we cannot add an event handler to

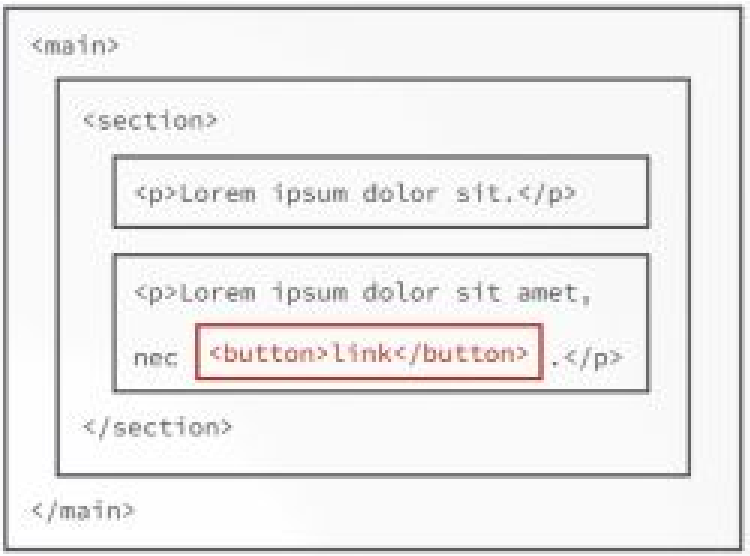
something that's not on our page, so in a case of deprecation that we're coding.

Suppose you have a list of 0, 10, or 100 items in the DOM when you load your page, and more items are waiting in your hand to add in the list. So there is no way to attach an event handler for the future elements or those elements are not added in the DOM yet, and also there may be a lot of items, so it wouldn't be useful to have one event handler attached to each of them.

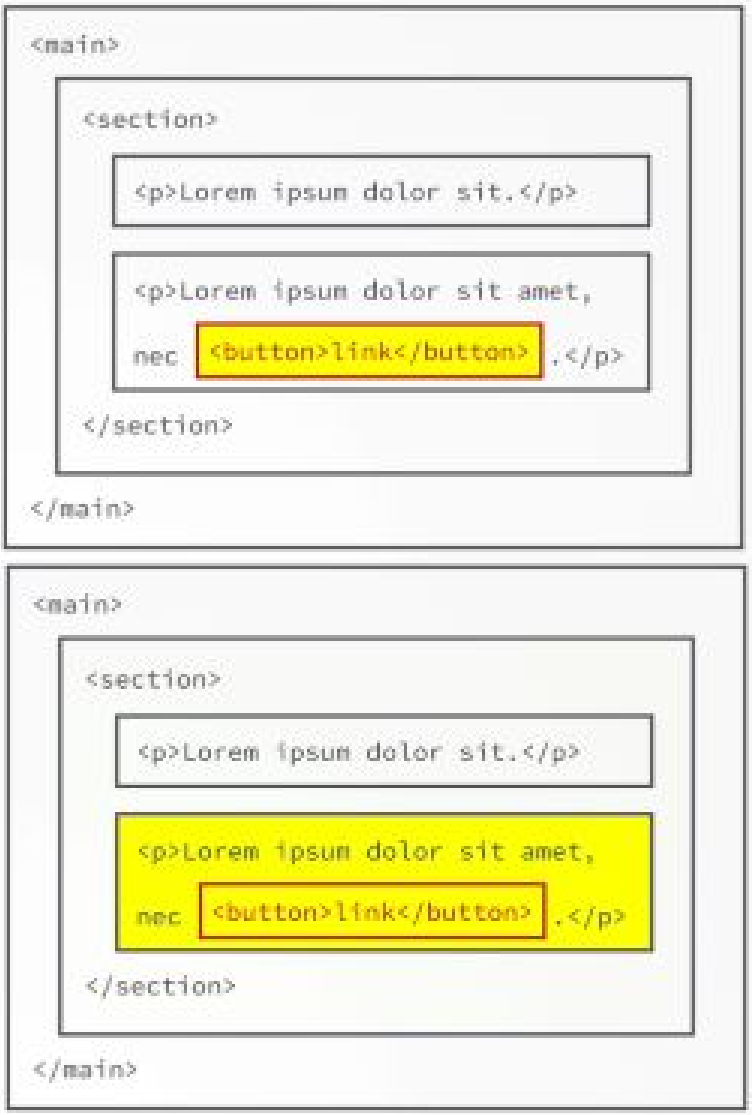
Event Delegation

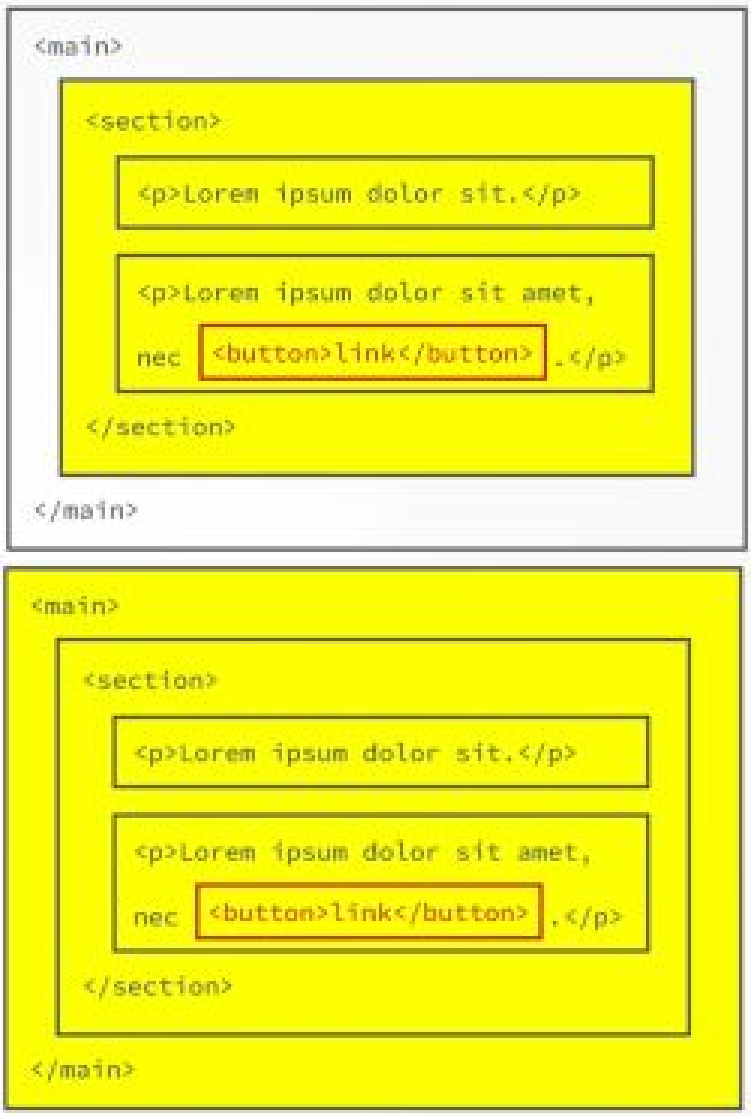
All right, so in order to talk about event delegation, the first concept that we actually need to talk about is event bubbling.

Event bubbling: Event bubbling means that when an event is fired or triggered on some DOM element, for example by clicking on our button here on the bellow image, then the exact same event is also triggered on all of the parent elements.



The event is first fired on the button, but then it will also be fired on all the parent elements one at a time, so it will also fire on the paragraph to the section the main element and actually all the way up in a DOM tree until the HTML element which is the root. So we say that the event bubbles up inside the DOM tree, and that's why it's called bubbling.





Target element: The element on which the event was actually first fired called the target element, so the element that caused the event to happen, is called the target element. In our above example here it's, of course, the button that was clicked. The important part is that this target element is stored as a property in the event object, This means that all the parent elements on which the event will also fire will know the target element of the event, so where the event was first fired.

That brings us to event delegation because if the event bubbles up in the DOM tree, and if we know where the event was fired then we can simply attach an event handler to a parent element and wait for the event to

bubble up, and we can then do whatever we intended to do with our target element. This technique is called event delegation. In this example here, we could simply add the event handler to the main element.

All right, so again, event delegation is to not set up the event handler on the original element that we're interested in but to attach it to a parent element and, basically, catch the event there because it bubbles up. We can then act on the element that we're interested in using the target element property.

Example: Now lets assume we have two list item in our page, after adding items in those list programmtically we want to delete one or more items from them. Using event delegation tecnique we can achive our ppurpose easily.

```
<div class="body">
    <div class="top">

    </div>
    <div class="bottom">
        <div class="other">
            <!-- other bottom elements -->
        </div>
        <div class="container clearfix">
            <div class="income">
                <h2 class="icome__title">Income</h2>
                <div class="income__list">
                    <!-- list items -->
                </div>
            </div>
            <div class="expenses">
                <h2 class="expenses__title">Expenses</
                <div class="expenses__list">
                    <!-- list items -->
                </div>
```

```
            </div>
        </div>
    </div>
</div>
```

Adding items in those list:

```
const DOMstrings={
        type:{
            income:'inc',
            expense:'exp'
        },
        incomeContainer:'.income__list',
        expenseContainer:'.expenses__list',
        container:'.container'
   }


var addListItem = function(obj, type){
        //create html string with the place holder
        var html, element;
        if(type===DOMstrings.type.income){
            element = DOMstrings.incomeContainer
            html = `<div class="item clearfix" id="inc
            <div class="item__description">${obj.descr
            <div class="right clearfix">
                <div class="item__value">${obj.value}<
                <div class="item__delete">
                    <button class="item__delete--btn">
outline"></i></button>
                </div>
            </div>
        </div>`
        }else if (type ===DOMstrings.type.expense){
            element=DOMstrings.expenseContainer;
            html = ` <div class="item clearfix" id="ex
            <div class="item__description">${obj.descr
            <div class="right clearfix">
                <div class="item__value">${obj.value}<
                <div class="item__percentage">21%</div
                <div class="item__delete">
                    <button class="item__delete--btn">
```

```
outline"></i></button>
                </div>
            </div>
        </div>`
        }
        var htmlObject = document.createElement('div')
        htmlObject.innerHTML=html;
        document.querySelector(element).insertAdjacent
htmlObject);
    }
```

Delete items:

```
var ctrlDeleteItem = function(event){
       // var itemId =
event.target.parentNode.parentNode.parentNode.parentNo
        var parent = event.target.parentNode;
        var splitId, type, ID;
        while(parent.id===""){
            parent = parent.parentNode
        }
        if(parent.id){
            splitId = parent.id.split('-');
            type = splitId[0];
            ID=parseInt(splitId[1]);
        }

        deleteItem(type, ID);
        deleteListItem(parent.id);
 }

 var deleteItem = function(type, id){
        var ids, index;
        ids = data.allItems[type].map(function(current
            return current.id;
        });
        index = ids.indexOf(id);
        if(index>-1){
            data.allItems[type].splice(index,1);
        }
    }
```

```
var deleteListItem = function(selectorID){
      var element = document.getElementById(selector
      element.parentNode.removeChild(element);
  }
```

Share Improve this answer Follow

edited May 4, 2020 at 8:23

answered May 4, 2020 at 4:16

Rafiq

11.3k ● 5 ● 42 ● 44

8

Event delegation is handling an event that *bubbles* using an event handler on a container element, but only activating the event handler's behavior if the event happened on an element within the container that matches a given condition. This can simplify handling events on elements within the container.

For instance, suppose you want to handle a click on any table cell in a big table. You *could* write a loop to hook up a click handler to each cell...or you could hook up a click handler on the table and use event delegation to trigger it only for table cells (and not table headers, or the whitespace within a row around cells, etc.).

It's also useful when you're going to be adding and removing elements from the container, because you don't have to worry about adding and removing event handlers on those elements; just hook the event on the container and handle the event when it bubbles.

Here's a simple example (it's intentionally verbose to allow for inline explanation): Handling a click on any `td` element in a container table:

```
// Handle the event on the container
document.getElementById("container").addEventListene
{
    // Find out if the event targeted or bubbled thr
this container element
    var element = event.target;
    var target;
    while (element && !target) {
        if (element.matches("td")) {
            // Found a `td` within the container!
            target = element;
        } else {
            // Not found
            if (element === this) {
                // We've reached the container, stop
                element = null;
            } else {
                // Go to the next parent in the ance
                element = element.parentNode;
            }
        }
    }
    if (target) {
        console.log("You clicked a td: " + target.te
    } else {
        console.log("That wasn't a td in the contain
    }
});
```

```
table {
    border-collapse: collapse;
    border: 1px solid #ddd;
}
th, td {
    padding: 4px;
```

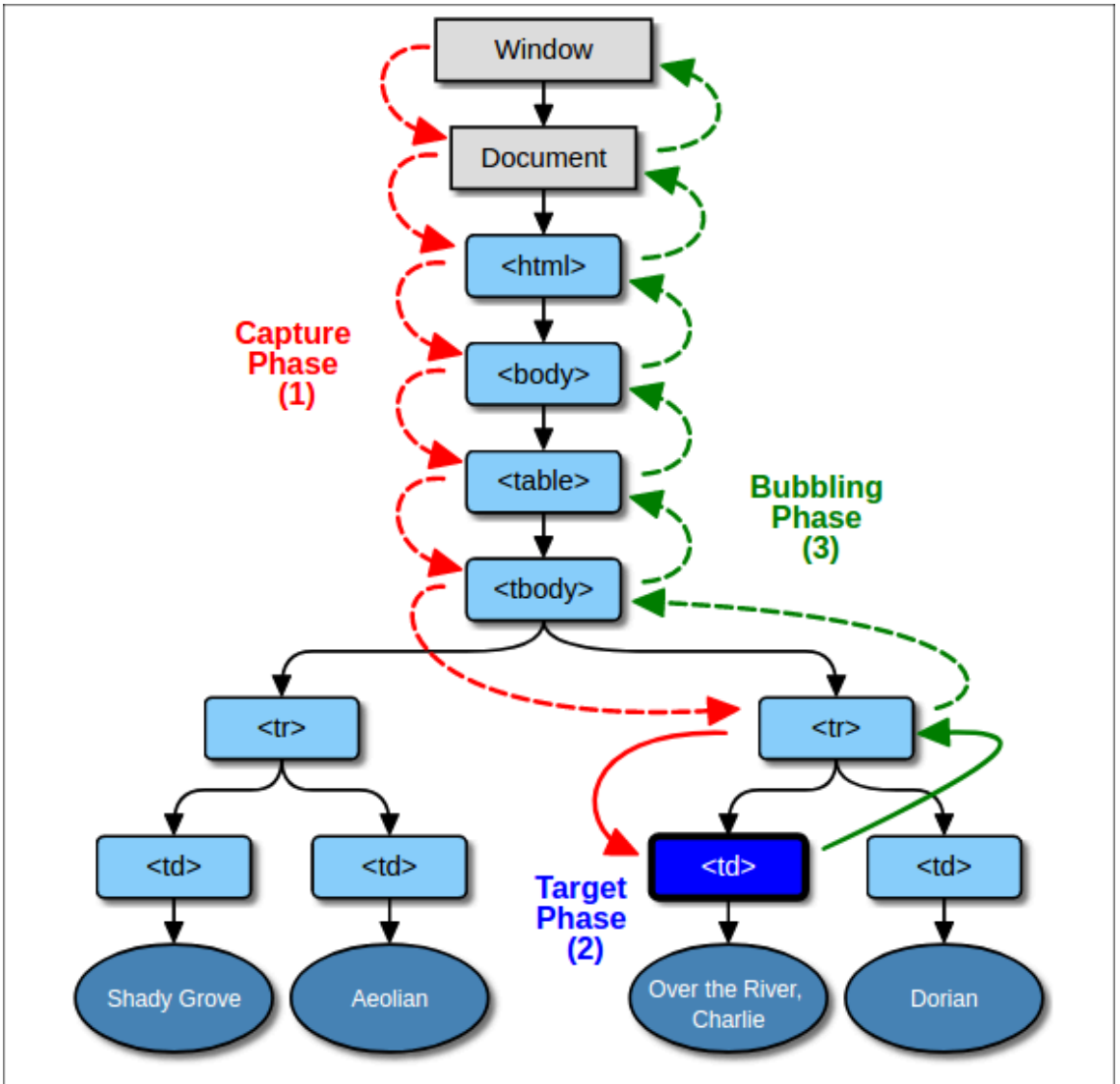
```
    border: 1px solid #ddd;
    font-weight: normal;
}
th.rowheader {
    text-align: left;
}
td {
    cursor: pointer;
}
```

```
<table id="container">
    <thead>
        <tr>
            <th>Language</th>
            <th>1</th>
            <th>2</th>
            <th>3</th>
        </tr>
    </thead>
    <tbody>
        <tr>
            <th class="rowheader">English</th>
            <td>one</td>
            <td>two</td>
            <td>three</td>
        </tr>
        <tr>
            <th class="rowheader">Español</th>
            <td>uno</td>
            <td>dos</td>
            <td>tres</td>
        </tr>
        <tr>
            <th class="rowheader">Italiano</th>
            <td>uno</td>
            <td>due</td>
            <td>tre</td>
        </tr>
    </tbody>
</table>
```

Run code snippet Expand snippet

Before going into the details of that, let's remind ourselves how DOM events work.

DOM events are dispatched from the document to the target element (the *capturing* phase), and then bubble from the target element back to the document (the *bubbling* phase). This graphic in the old DOM3 events spec (now superceded, but the graphic's still valid) shows it really well:



Graphical representation of an event dispatched in a DOM tree using the DOM event flow

Not all events bubble, but most do, including `click`.

The comments in the code example above describe how it works. `matches` checks to see if an element matches a CSS selector, but of course you can check for whether something matches your criteria in other ways if you don't want to use a CSS selector.

That code is written to call out the individual steps verbosely, but on vaguely-modern browsers (and also on IE if you use a polyfill), you can use `closest` and `contains` instead of the loop:

```
var target = event.target.closest("td");
    console.log("You clicked a td: " + target.textCont
} else {
    console.log("That wasn't a td in the container tab
}
```

Live Example:

▶ Show code snippet

`closest` checks the element you call it on to see if it matches the given CSS selector and, if it does, returns that same element; if not, it checks the parent element to see if it matches, and returns the parent if so; if not, it checks the parent's parent, etc. So it finds the "closest" element in the ancestor list that matches the selector. Since that might go past the container element, the code above uses `contains` to check that if a matching element was found, it's within the container — since by hooking

the event on the container, you've indicated you only want to handle elements *within* that container.

Going back to our table example, that means that if you have a table within a table cell, it won't match the table cell containing the table:

Show code snippet

Share Improve this answer

Follow

answered Apr 1, 2019 at 10:25

T.J. Crowder

1.1m ● 199 ● 2k ● 1.9k

7

The delegation concept

If there are many elements inside one parent, and you want to handle events on them of them - don’t bind handlers to each element. Instead, bind the single handler to their parent, and get the child from event.target. This site provides useful info about how to implement event delegation.
http://javascript.info/tutorial/event-delegation

Share Improve this answer

Follow

edited Jun 20, 2020 at 9:12

Community Bot

1 ● 1

answered Jan 1, 2014 at 7:28

Joseph

91 ● 1 ● 7

6

Delegation is a technique where an object expresses certain behavior to the outside but in reality delegates responsibility for implementing that behaviour to an associated object. This sounds at first very similar to the proxy pattern, but it serves a much different purpose. Delegation is an abstraction mechanism which centralizes object (method) behavior.

Generally spoken: use delegation as alternative to inheritance. Inheritance is a good strategy, when a close relationship exist in between parent and child object, however, inheritance couples objects very closely. Often, delegation is the more flexible way to express a relationship between classes.

This pattern is also known as "proxy chains". Several other design patterns use delegation - the State, Strategy and Visitor Patterns depend on it.

Share Improve this answer

Follow

answered Nov 6, 2009 at 12:40

Ewan Todd

7,347 ● 28 ● 35

1 Good explanation. In the example of the <ul> with several <li> children, apparently the <li> are those that handle the click logic, but it is not like that because they "delegate" this logic in the father <ul> – Juanma Menendez Jan 15, 2019 at 0:43

2

Event delegation makes use of two often overlooked features of JavaScript events: event bubbling and the target element.When an event is triggered on an element, for example a mouse click on a button, the same event is also triggered on all of that element's ancestors. This process is known as event bubbling; the event bubbles up from the originating element to the top of the DOM tree.

Imagine an HTML table with 10 columns and 100 rows in which you want something to happen when the user clicks on a table cell. For example, I once had to make each cell of a table of that size editable when clicked. Adding event handlers to each of the 1000 cells would be a major performance problem and, potentially, a source of browser-crashing memory leaks. Instead, using event delegation, you would add only one event handler to the table element, intercept the click event and determine which cell was clicked.

Share Improve this answer

Follow

answered May 1, 2019 at 8:39

Priti jha

143 ● 1 ● 4

2

It's basically how association is made to the element. `.click` applies to the current DOM, while `.on` (using delegation) will continue to be valid for new elements added to the DOM after event association.

Which is better to use, I'd say it depends on the case.

Example:

```
<ul id="todo">
   <li>Do 1</li>
   <li>Do 2</li>
   <li>Do 3</li>
   <li>Do 4</li>
</ul>
```

.Click Event:

```
$("li").click(function () {
   $(this).remove ();
});
```

Event .on:

```
$("#todo").on("click", "li", function () {
   $(this).remove();
});
```

Note that I've separated the selector in the .on. I'll explain why.

Let us suppose that after this association, let us do the following:

```
$("#todo").append("<li>Do 5</li>");
```

That is where you will notice the difference.

If the event was associated via .click, task 5 will not obey the click event, and so it will not be removed.

If it was associated via .on, with the selector separate, it will obey.

Share Improve this answer

Follow

edited Jan 13, 2023 at 8:35

dumbass

27.2k ● 4 ● 35 ● 72

answered Sep 13, 2018 at 19:01

Lucas

301 ● 8 ● 37

This answer is wrong. According to the jQuery docs for on and click both behave the same, the latter is only a shorthand. In your code example you do select the targets differently and therefore you get different behavior. As an aside your answer is specifically dealing with jQuery but this fact is never mentioned. – Martin Jul 28, 2022 at 21:01

0

Event Delegation

Attach an event listener to a parent element that fires when an event occurs on a child element.

Event Propagation

When an event moves through the DOM from child to a parent element, that's called *Event Propagation*, because the event propagates, or moves through the DOM.

In this example, an event (onclick) from a button gets passed to the parent paragraph.

```
$(document).ready(function() {

    $(".spoiler span").hide();

    /* add event onclick on parent (.spoiler) and de
(button) */
    $(".spoiler").on( "click", "button", function()

        $(".spoiler button").hide();

        $(".spoiler span").show();

    } );

});
```

```
<script src="https://ajax.googleapis.com/ajax/libs/j
</script>

<p class="spoiler">
    <span>Hello World</span>
    <button>Click Me</button>
</p>
```

Run code snippet Expand snippet

Codepen

Share Improve this answer

Follow

edited Sep 14, 2019 at 13:32

answered Sep 14, 2019 at 13:19

antelove

3,348 ● 28 ● 20