



WEB TECHNOLOGIES 1

Events and Event Handler

EVENT

- *An event* is a signal that something has happened. All DOM nodes generate such signals.
- *Events* are actions that occur, usually as a result of something the user does.
- For example,

This can be the end user clicking on something, moving the mouse over a certain element or pressing down certain keys on the keyboard.

- An event can also be something that happens in the web browser, such as the web page completing the loading of a page, or the user scrolling or resizing the window.
- *Event handlers*, scripts that are automatically executed when an event occurs.

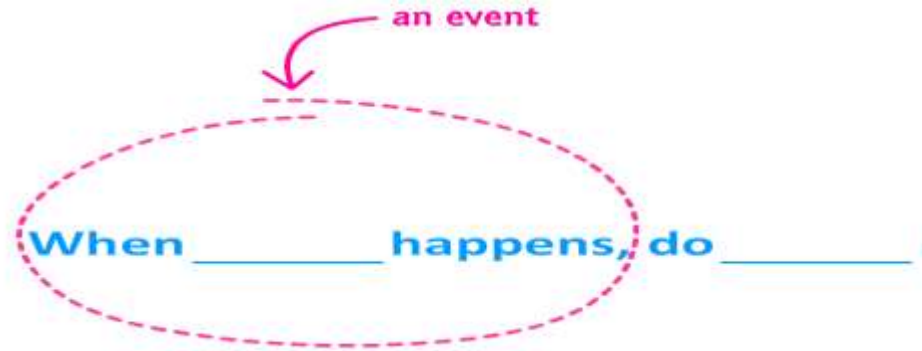


WHAT ARE EVENTS?

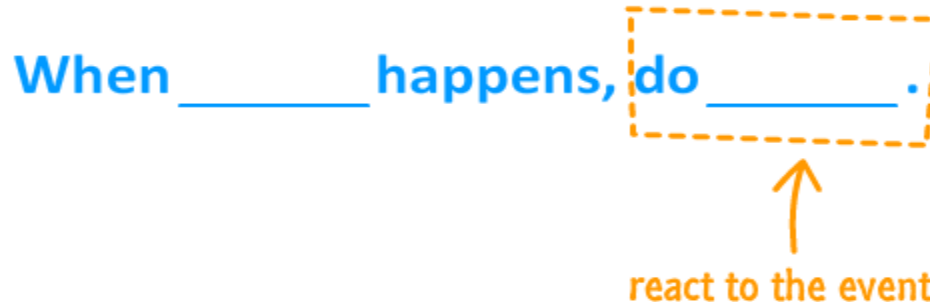
- An event is nothing more than a signal.
- Its an object that is implicitly created by the browser .
- It communicates that something has just happened. This something could be a mouse click.
- To detect the activities between the browser and the user is known as events and to provide computation when these occurs is specified through event-driven programming.
- **Event-driven programming** is the part of program executed at completely unpredictable moment depending on user interaction triggers.



WHAT ARE EVENTS?



- An event defines the thing that happens. **It fires the signal.**
- The second part of the model is defined by the reaction to the event.
- The first blank calls out something that happens. The second blank describes the reaction to that.



WHAT ARE EVENT HANDLER?

- An *event handler* is a script that is implicitly executed in response to the appearance of an event.
- Event handlers enable a Web document to be responsive to browser and user activities.
- Event handlers are used to check for simple errors and omissions in user input in the form, either when they are changed or when the form is submitted.
- This saves the time of sending incorrect form data to the server.



EVENT AND EVENT HANDLER

- Events are created by activities associated with specific HTML elements.
 - For example, the click event can be caused by the browser user clicking a radio button or the link of an anchor tag.
- The process of connecting an event handler to an event is called *registration*.
- There are two distinct approaches to event handler registration,
 - one that assigns tag attributes
 - one that assigns handler addresses to object properties.



WORK WITH EVENTS

- To work with events, there are two things we need to do:
 - Listen for events
 - React to events
- EventTarget is a DOM interface implemented by objects that can receive events and may have listeners for them.
- Element, Document and Window are the most common event targets, but other objects can be event targets too.
- Many event targets (including elements, documents, and windows) also support setting event handlers via *onevent* properties and attributes.



WORK WITH EVENTS

- Events are JavaScript objects, their names are case sensitive. The names of all event objects have only lowercase letters.
 - **For example, click is an event, but Click is not.**
- To work with events, there are two things you need to do:
 - Listen for events
 - inline model
 - traditional model
 - W3C model
 - React to events
 - Listening to events is handled by `addEventListener`. What to do after an event is overheard is handled by the event handler.




INLINE MODEL

- 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.

```
<!doctype html>
<title>HTML Attribute</title>
<body>
  <input value="Click me" onclick="alert('Click!')" type="button">
</body>
```

INLINE MODEL

```
<body>
<input type="button" id="askme" value="Ask me" onclick="ask()"/>
<script>
  function ask(){
    var a =window.prompt("Please ask a question")
    alert("I am sorry I don't know the answer")
  }
</script>
</body>
```



DOM PROPERTIES

- We can assign a handler using a DOM property on<event>.
- For instance, **elem.onclick**.
- 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.
- The handler is always in the DOM property: the HTML-attribute is just one of the ways to initialize it.



TRADITIONAL MODEL

```
<body>
  <input type="button" id="askme" value="Ask me"/>
  <script>
    var a=document.getElementById("askme");
    a.onclick=ask;
    function ask(){
      var a =window.prompt("Please ask a question")
      alert("I am sorry I don't know the answer")
    }
  </script>
</body>
```



W3C MODEL

```
<body>
  <input type="button" id="askme" value="Ask me" />
  <script>
    var a=document.getElementById("askme");
    a.addEventListener("click",ask,false)
    function ask(){
      var a =window.prompt("Please ask a question")
      alert("I am sorry I don't know the answer") }
  </script>
</body>
```



POSSIBLE MISTAKES

- The function should be assigned as `sayThanks`, not `sayThanks()`.

- Example:

```
button.onclick = sayThanks; // right
```

```
button.onclick = sayThanks(); // wrong
```

If parentheses is added, `sayThanks()` – is a function call.

- **Use functions, not strings.**

- The assignment `elem.onclick = "alert(1)"` would work too. It works for compatibility reasons, but not recommended

- **DOM-property case matters.**

- Assign a handler to `elem.onclick`, not `elem.ONCLICK`, because DOM properties are case-sensitive.

- **Don't use `setAttribute` for handlers.** Such a call won't work.



WORK WITH EVENTS

- `EventTarget()`

Creates a new `EventTarget` object instance.

- `EventTarget.addEventListener()`

Registers an event handler of a specific event type on the `EventTarget`.

- `EventTarget.removeEventListener()`

Removes an event listener from the `EventTarget`.



LISTENING FOR EVENTS

- `source.addEventListener(eventName, eventHandler, false);`

- *Source*

We call `addEventListener` via an element or object that we want to listen for events on. It will be a DOM element, but it can also be our document, window, or any object specially designed to fire events.

- *Event Name*

The first argument we specify to the `addEventListener` function is the name of the event we are interested in listening to.

- *Event Handler*

The second argument requires us to specify a function that will get called when the event gets overheard.

- *To Capture, or Not to Capture*

The last argument is made up of either a true or a false.



EXAMPLE

- `document.addEventListener("click",changeColor, false);`
- In the above example :
 - Document - is the source
 - “click” - is the name of the event that we are interested to listen
 - `changeColor` - is the Event handler function that will be called once the above event has occurred.
 - `false` - Boolean value specifies the Capture is disabled.



HTML/DOM EVENTS FOR JAVASCRIPT

| Events | Description |
|-------------|---|
| onclick | occurs when element is clicked. |
| ondblclick | occurs when element is double-clicked. |
| onfocus | occurs when an element gets focus such as button, input, textarea etc. |
| onblur | occurs when form loses the focus from an element. |
| onsubmit | occurs when form is submitted. |
| onmouseover | occurs when mouse is moved over an element. |
| onmouseout | occurs when mouse is moved out from an element (after moved over). |
| onmousedown | occurs when mouse button is pressed over an element. |
| onmouseup | occurs when mouse is released from an element (after mouse is pressed). |
| onload | occurs when document, object or frameset is loaded. |
| onunload | occurs when body or frameset is unloaded. |
| onscroll | occurs when document is scrolled. |

HTML/DOM EVENTS FOR JAVASCRIPT

| Events | Description |
|------------|-----------------------------------|
| onreset | occurs when form is reset. |
| onkeydown | occurs when key is being pressed. |
| onkeypress | occurs when user presses the key. |
| onkeyup | occurs when key is released. |

Always define functions for your event handlers because:

It makes your code modular - you can use the same function as an event handler for many different items.

It makes your code easier to read.



EVENT HANDLERS

- Events apply to HTML tags as follows:
 - Focus, Blur, Change events: text fields, textareas, and selections
- **Click events:** buttons, radio buttons, checkboxes, submit buttons, reset buttons, links
- **Select events:** text fields, textareas
- **MouseOver event:** links
- If an event applies to an HTML tag, then you can define an event handler for it. In general, an event handler has the name of the event, preceded by "on."
- For example, the event handler for the Focus event is **onFocus**.



EVENT HANDLERS

- Many objects also have methods that emulate events.
- For example, button has a click method that emulates the button being clicked.
- **Note:** The event-emulation methods do not trigger event-handlers. So, for example, the click method does not trigger an onClick event-handler.

| Event | Occurs when... | Event Handler |
|-----------|---|---------------|
| blur | User removes input focus from form element | onBlur |
| click | User clicks on form element or link | onClick |
| change | User changes value of text, textarea, or select element | onChange |
| focus | User gives form element input focus | onFocus |
| load | User loads the page in the Navigator | onLoad |
| mouseover | User moves mouse pointer over a link or anchor | onMouseOver |
| select | User selects form element's input field | onSelect |
| submit | User submits a form | onSubmit |
| unload | User exits the page | onUnload |



EVENT HANDLERS

- HTML attributes are used sparingly, because JavaScript in the middle of an HTML tag looks a little bit odd and alien. Also can't write lots of code in there.
- DOM properties are ok to use, but we can't assign more than one handler of the particular event. In many cases that limitation is not pressing.
- The last way is the most flexible, but it is also the longest to write.
- There are few events that only work with it, for instance `transitionend` and `DOMContentLoaded`. Also `addEventListener` supports objects as event handlers. In that case the method `handleEvent` is called in case of the event.



ADDEVENTLISTENER()

- It sets up a function that will be called whenever the specified event is delivered to the target.
- The first parameter is the type of the event (like "click" or "mousedown" or any other HTML DOM Event.)
- The second parameter is the function we want to call when the event occurs.
- The third parameter is a boolean value specifying whether to use event bubbling or event capturing. This parameter is optional.

```
<body>
  <script>
    document.addEventListener("click", changeColor, false);

    function changeColor() {
      document.body.style.backgroundColor = "blue";
    }
  </script>
</body>
</html>
```

ADDEVENTLISTENER()

- *addEventListener()* is the way to register an event listener as specified in W3C DOM. The benefits are as follows:
 - It allows adding more than a single handler for an event. This is particularly useful for AJAX libraries, JavaScript modules, or any other kind of code that needs to work well with other libraries/extensions.
 - It gives you finer-grained control of the phase when the listener is activated (capturing vs. bubbling).
 - It works on any DOM element, not just HTML elements.
 - If multiple identical EventListeners are registered on the same EventTarget with the same parameters, the duplicate instances are discarded. They do not cause the EventListener to be called twice.



REMOVEEVENTLISTENER()

- The **EventTarget.removeEventListener()** method removes from the EventTarget an event listener previously registered.
- **Syntax:**
 - `element.removeEventListener(event, listener, useCapture)`
- **Parameters:** It accepts three parameters which are specified below-
- **event:** It is a string which describes the name of event that has to be remove.
- **listener:** It is the function of the event handler to remove.
- **useCapture:** It is an optional parameter. By default it is Boolean value false which specifies the removal of event handler from the bubbling phase and if it is true than the removeEventListener() method removes the event handler from the capturing phase.



LOAD & ONCLICK

- The onload event in JavaScript triggers when a web page loads in a browser.
- The onload Event is defined in BODY element.
- The onclick event in JavaScript simply triggers when you click a specific control, such as button control.



EVENTOBJECT & THIS

- When a event listener's event occurs and it calls its associated function, it also passes a single argument to the function—a reference to the event object.
- The event object contains a number of properties that describe the event that occurred.

| W3C Name | Microsoft Name | Description |
|--|---------------------------|--|
| <code>e</code> | <code>window.event</code> | The object containing the event properties |
| <code>type</code> | <code>type</code> | The event that occurred (click, focus, blur, etc.) |
| <code>target</code> | <code>srcElement</code> | The element to which the event occurred |
| <code>keyCode</code> | <code>keyCode</code> | The numerical ASCII value of the pressed key |
| <code>shiftKey</code> <code>altKey</code> <code>ctrlKey</code> | | Returns 1 if pressed, 0 if not |
| <code>currentTarget</code> | <code>fromElement</code> | The element the mouse came from on mouseover |
| <code>relatedTarget</code> | <code>toElement</code> | The element the mouse went to on mouseout |



EVENTOBJECT & THIS

- By convention, the parameter name *e* is used in event-triggered functions to receive the event object argument. If I wanted to determine the type of event that occurred, such as a click, example:
- ```
function myEvent(e)

{

 var evtType = e.type alert(evtType) // displays click, or whatever the
 event type was

}
```
- In JavaScript the *this* keyword always refers to the “owner” of a function. hence event handlers it is very useful if this refers to the HTML element the event is handled by, so that you have easy access to it.
- In the W3C model it works the same as in the traditional model , it refers to the HTML element currently handling the event.



# EVENTOBJECT & THIS

- If you register `doSomething()` as the click event handler of any HTML element, that element gets a red background whenever the user clicks on it.

- `element.addEventListener('click',doSomething,false);`  
`another_element.addEventListener('click',doSomething,false);`

```
function doSomething()
```

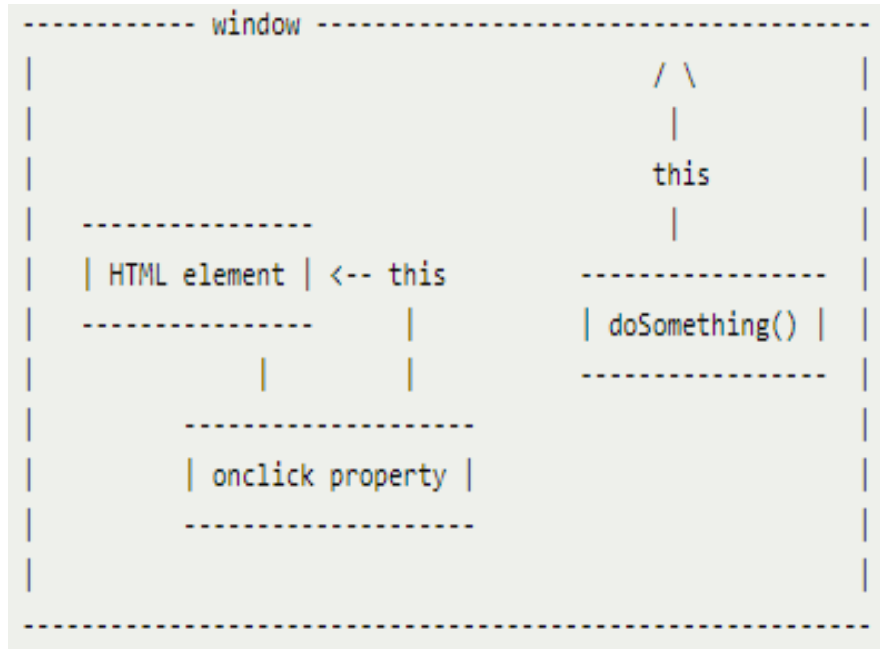
```
{
```

```
 this.style.backgroundColor = '#cc0000';
```

```
}
```



# EVENTOBJECT & THIS

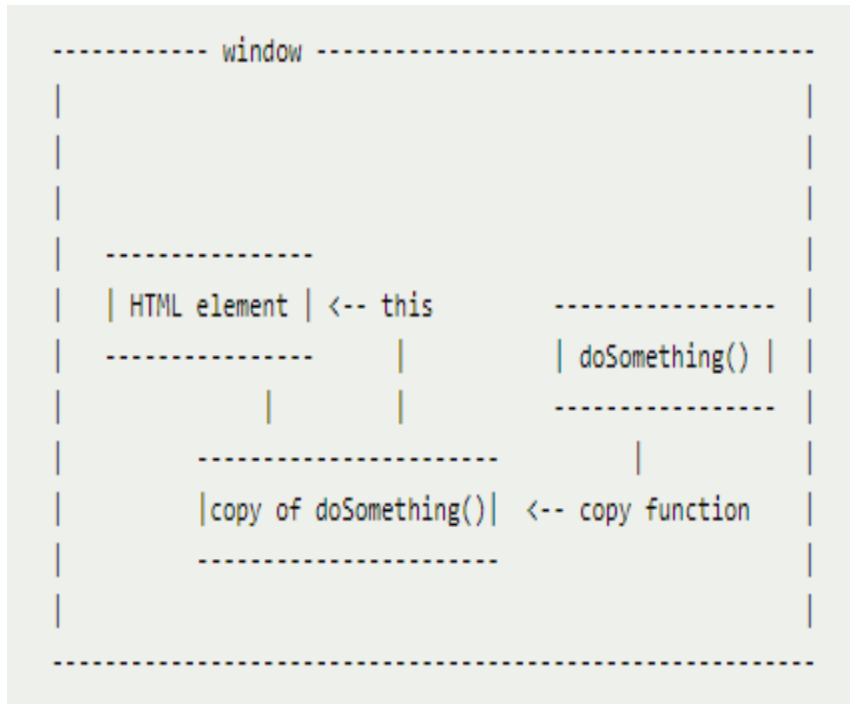


In JavaScript *this* always refers to the “owner” of the function we're executing, or rather, to the object that a function is a method of.

The function `doSomething()` in a page, its *owner* is the page, or rather, the window object (or global object) of JavaScript.

An `onclick` property, though, is owned by the HTML element it belongs to.

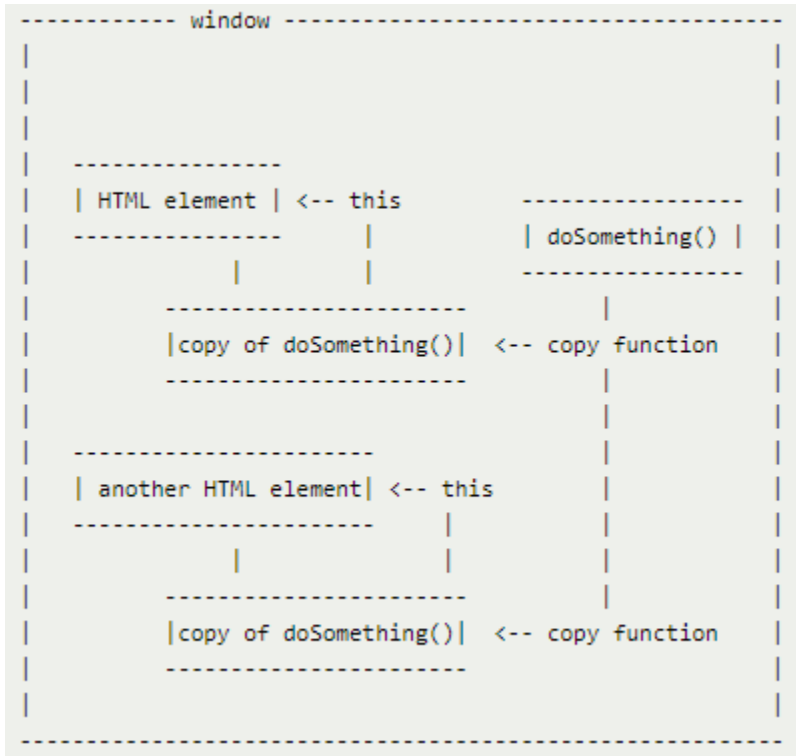
# EVENTOBJECT & THIS



The function is copied in its entirety to the `onclick` property (which now becomes a method). So if the event handler is executed, `this` refers to the HTML element and its color is changed.



# EVENTOBJECT & THIS

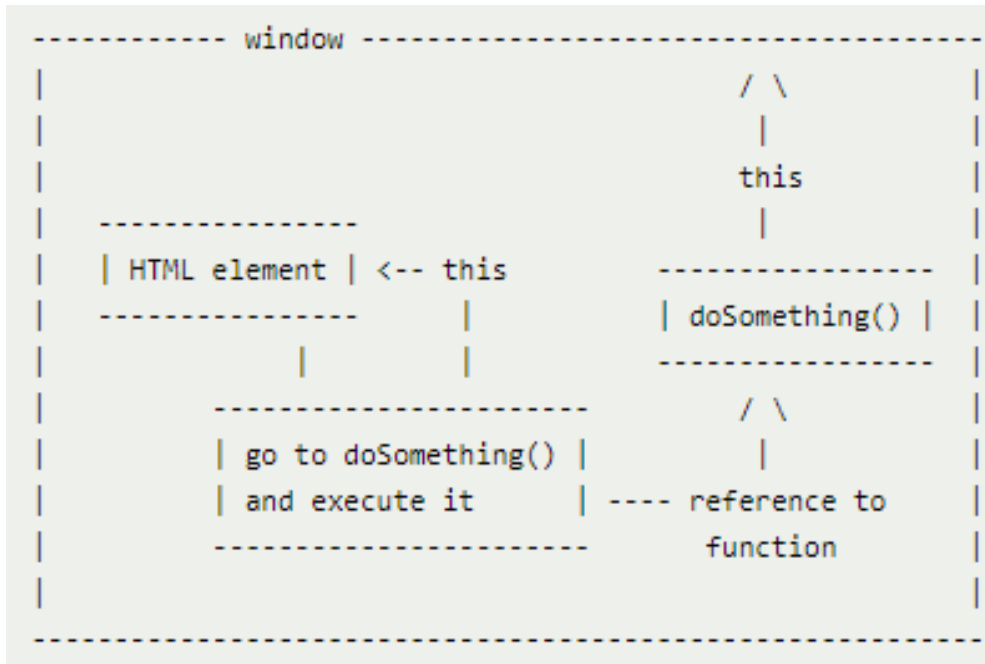


The use of `this` is fullest extent. Each time the function is called, `this` refers to the HTML element that is currently handling the event, the HTML element that "owns" the copy of `doSomething()`.





# EVENTOBJECT & THIS



If you refer to it, the difference is crucial. The onclick property does not contain the actual function, but merely a function call.

When we arrive at doSomething() the this keyword once again refers to the global window object and the function returns error messages.

# EVENTOBJECT & THIS

## Examples - copying

this is written into the onclick method in the following cases:

```
element.onclick = doSomething
element.addEventListener('click',doSomething,false)
element.onclick = function () {this.style.color = '#cc0000';}
<element onclick="this.style.color = '#cc0000';">
```

## Examples - referring

In the following cases this refers to the window:

```
element.onclick = function () {doSomething()}
element.attachEvent('onclick',doSomething)
<element onclick="doSomething()">
```



# EVENT OBJECT PROPERTIES

| Property                                      | Description                                                                                                                                                         |
|-----------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>altKey</code>                           | This value is <b>true</b> if the <i>Alt</i> key was pressed when the event fired.                                                                                   |
| <code>cancelBubble</code>                     | Set to <b>true</b> to prevent the event from bubbling. Defaults to <b>false</b> . (See Section 14.9, Event Bubbling.)                                               |
| <code>clientX</code> and <code>clientY</code> | The coordinates of the mouse cursor inside the client area (i.e., the active area where the web page is displayed, excluding scrollbars, navigation buttons, etc.). |
| <code>ctrlKey</code>                          | This value is <b>true</b> if the <i>Ctrl</i> key was pressed when the event fired.                                                                                  |
| <code>keyCode</code>                          | The ASCII code of the key pressed in a keyboard event. See Appendix D for more information on the ASCII character set.                                              |
| <code>screenX</code> and <code>screenY</code> | The coordinates of the mouse cursor on the screen coordinate system.                                                                                                |
| <code>shiftKey</code>                         | This value is <b>true</b> if the <i>Shift</i> key was pressed when the event fired.                                                                                 |
| <code>type</code>                             | The name of the event that fired, without the prefix "on".                                                                                                          |



# ONSUBMIT, ONRESET, ONFOCUS, ONBLUR

- The *onsubmit* event in JavaScript is used with FORM element and is triggered when the form is submitted.
- The *onreset* event in JavaScript is defined with a form and is triggered when the fields of form are reset.
- The *onblur* event occurs when an object loses focus. The *onblur* event is most often used with form validation code (e.g. when the user leaves a form field).
- The *onfocus* event occurs when an element gets focus. The *onfocus* event is most often used with <input>, <select>, and <a>.
- The *onfocus* event is the opposite of the *onblur* event.



# MOUSE EVENTS

- The mouse events in JavaScript are those events that triggers while operating the mouse.
- **click** – when the mouse clicks on an element (touchscreen devices generate it on a tap).
- **contextmenu** – when the mouse right-clicks on an element.
- **mousedown** / **mouseup** – when the mouse button is pressed / released over an element.
- **mousemove** – when the mouse is moved.



# ONMOUSEOVER AND ONMOUSEOUT

- The mouseover event occurs when a mouse pointer comes over an element, and mouseout – when it leaves.
- A fast mouse move may skip intermediate elements.
- They are all about hovering over something and hovering away from something.

- ONMOUSEOVER :

This event corresponds to hovering the mouse pointer over the element and its children, to which it is bound to.

- ONMOUSEOUT :

Whenever the mouse cursor leaves the element which handles a mouseout event, a function associated with it is executed.



# MOUSEENTER AND MOUSELEAVE

- The mouseenter and mouseleave events do not bubble. This makes them unique.
- All four of these events behave identically when there are no child elements at play. If there are child elements at play :
  - *mouseover* and *mouseout* will get fired each time you move the mouse over and around a child element. This means that you could be seeing many unnecessary event fires even though it seems like you are moving your mouse within a single region.
  - *mouseenter* and *mouseleave* will get fired only once. It doesn't matter how many child elements your mouse moves through.



# EVENT 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.
- In cases when a single action initiates multiple events, their order is fixed. That is, the handlers are called in the order

**mousedown → mouseup → click**





# EVENT ORDER

- Click-related events always have the `which` property, which allows to get the exact mouse button.
- It is not used for `click` and `contextmenu` events, because the former happens only on left-click, and the latter – only on right-click.
- But if we track `mousedown` and `mouseup`, then we need it, because these events trigger on any button, so `which` allows to distinguish between “right-mousedown” and “left-mousedown”.
- There are the three possible values:
  - `event.which == 1` – the left button
  - `event.which == 2` – the middle button
  - `event.which == 3` – the right button



# MODIFIERS :SHIFT,ALT,CTRL,META

- All mouse events include the information about pressed modifier keys.
- Event properties:
  - `shiftKey`: Shift
  - `altKey`: Alt (or Opt for Mac)
  - `ctrlKey`: Ctrl
  - `metaKey`: Cmd for Mac
- They are true if the corresponding key was pressed during the event.



# COORDINATES

- `screenX`, `screenY`, `clientX`, and `clientY` returns a number which indicates the number of physical “CSS pixels” a point is from the reference point.
- The event point is where the user clicked, the reference point is a point in the upper left.
- These properties return the horizontal and vertical distance from that reference point.
- **`clientX/Y`** gives the coordinates relative to the viewport in CSS pixels.
- **`screenX/Y`** gives the coordinates relative to the screen in device pixels.



# MOUSEEVENT CLIENTX & CLIENTY PROPERTY

- MouseEvent clientX Property
  - The clientX property returns the horizontal coordinate (according to the client area) of the mouse pointer when a mouse event was triggered.
- MouseEvent clientY Property
  - The clientY property returns the vertical coordinate (according to the client area) of the mouse pointer when a mouse event was triggered.
- The client area is the current window.



# MOUSEEVENT SCREENX & SCREENY PROPERTY

- Gives coordinates of the mouse pointer, relative to the screen, when the mouse button is clicked on an element.
- MouseEvent screenX Property
  - The screenX property returns the horizontal coordinate (according to the users computer screen) of the mouse pointer when an event was triggered.
- MouseEvent screenY Property
  - The screenY property returns the vertical coordinate (according to the users computer screen) of the mouse pointer when an event was triggered.



# MOUSEMOVE EVENT

- The **mousemove event** occurs whenever the mouse pointer moves within the selected element.
- The **mousemove()** method triggers the **mousemove event**, or attaches a function to run when a **mousemove event** occurs.



# MOUSE EVENTS

- Dblclick :

- The dblclick event is fired when you basically quickly repeat a click action a double number of times.
- The amount of time between each click that ends up resulting in a dblclick event is based on the OS you are running the code in.
- It's neither browser specific nor something you can define (or read) using JavaScript.



# OTHER EVENTS

- **Form element events:**

- submit – when the visitor submits a `<form>`.
- focus – when the visitor focuses on an element, e.g. on an `<input>`.

- **Keyboard events:**

- keydown and keyup – when the visitor presses and then releases the button.

- **Document events:**

- DOMContentLoaded – when the HTML is loaded and processed, DOM is fully built.

- **CSS events:**

- transitionend – when a CSS-animation finishes.





# ONCONTEXTMENU

- It occurs when the right mouse button is clicked on an element and the *contextmenu* is shown.
- The *oncontextmenu* event is cancelable, if you cancel it, the context menu is not shown.
- The *contextmenu* event is fired just before this menu appears.
- It is used to prevent the menu from appearing when you right-click
- The *preventDefault* method on an Event stops whatever the default behavior is from actually happening.



# KEYBOARD EVENTS

- Here are three events and those events are
  - keydown
  - keypress
  - Keyup
- The *keydown* event is fired when you press down on a key on your keyboard.
- The *keyup* event is fired when you release a key that you just pressed. Both of these events work on any key that you interact with.



# KEYBOARD EVENT

- *keyCode*

Every key we press on your keyboard has a number associated with it. This read-only property returns that number.

- *charCode*

This property only exists on event arguments returned by the **keypress** event, and it contains the ASCII code for whatever character key you pressed.

- The `charCode` and `keyCode` values for a particular key are not the same.
- Also, the `charCode` is only returned if the event that triggered your event handler was `keypress`.



# KEYBOARD EVENTS

- The **keypress** event is fired only when you press down on a key that displays a character (letter, number, etc).
- If you press and release a character key such as the letter **y**, you will see the **keydown**, **keypress**, and **keyup** events fired in order. The **keydown** and **keyup** events fire because the **y** key is simply a key to them.
- The **keypress** event is fired because the **y** key is a character key.
- If you press and release a key that doesn't display anything on the screen all you will see are the **keydown** and **keyup** events fired.



# THE KEYBOARD EVENT PROPERTIES


## ○ *metaKey*

- The `metaKey` property is similar to the `ctrlKey`, `altKey`, and `shiftKey` properties in that it returns a **true** if the Meta key is pressed.
- The Meta key is the Windows key on Windows keyboards and the Command key on Apple keyboards.
- ASCII Code URL :

<https://www.cs.cmu.edu/~pattis/15-1XX/common/handouts/ascii.html>



# EVENT PROPAGATION

- When an event occurs in an element inside another element, and both elements have registered a handle for that event.
  - The term *event propagation* is often used as a synonym of *event bubbling*.
  - Event bubbling and capturing are two ways of event propagation in the HTML DOM API.
  - Propagation is the process of calling all the listeners for the given event type, attached to the nodes on the branch.
  - Each listener will be called with an event object that gathers information relevant to the event.
  - Several listeners can be registered on a node for the same event type.
  - When the propagation reaches one such node, listeners are invoked in the order of their registration.
- 

# EVENT PROPAGATION

- Event Flow process is completed by three concepts :
  - Event Capturing.
  - Event Target.
  - Event Bubbling.
- From the window to the event target parent: this is the *capture phase*
- The event target itself: this is the *target phase*
- From the event target parent back to the window: the *bubble phase*.




# EVENT FLOW

- Event flow is the *order* in which event is received on the web page.
- If you click on an element like on div or on the button , which is nested to other elements, before the click is performed on the target element.
- It must trigger the click event each of its parent elements first, starting at the top with the global window object.
- By default, *every element* of HTML is **child** of the *window object*.
- 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.





# EVENT BUBBLING

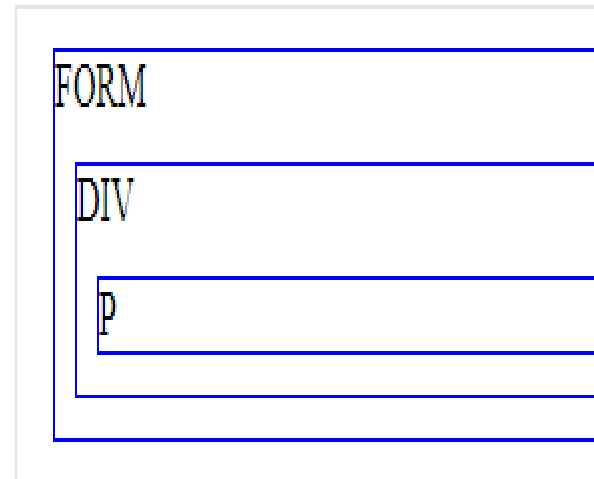
- The *event bubbling* is used where a single event, such as a mouse click, may be handled by two or more event handlers defined at different levels of the *Document Object Model* (DOM) hierarchy.
  - The event bubbling process starts by executing the event handler defined for individual elements at the lowest level (e.g. individual hyperlinks, buttons, table cells etc.).
  - From there, the event *bubbles up* to the containing elements (e.g. a table or a form with its own event handler), then up to even higher-level elements (e.g. the BODY element of the page).
  - Finally, the event ends up being handled at the highest level in the DOM hierarchy, the document element itself (provided that your document has its own event handler).
- 

# EVENT 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.
- Let's say we have 3 nested elements FORM > DIV > P with a handler on each of them:

```
<style>
 body * {
 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>
```

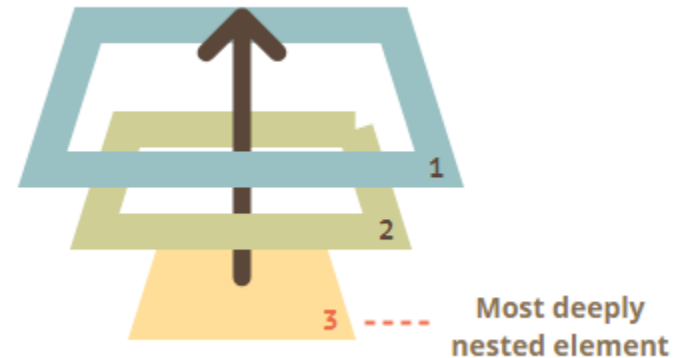


# EVENT BUBBLING

- A click on the inner `<p>` first runs onclick:
  - On that `<p>`.
  - Then on the outer `<div>`.
  - Then on the outer `<form>`.
- And so on upwards till the document object.
- So if we click on `<p>`, then we'll see 3 alerts:

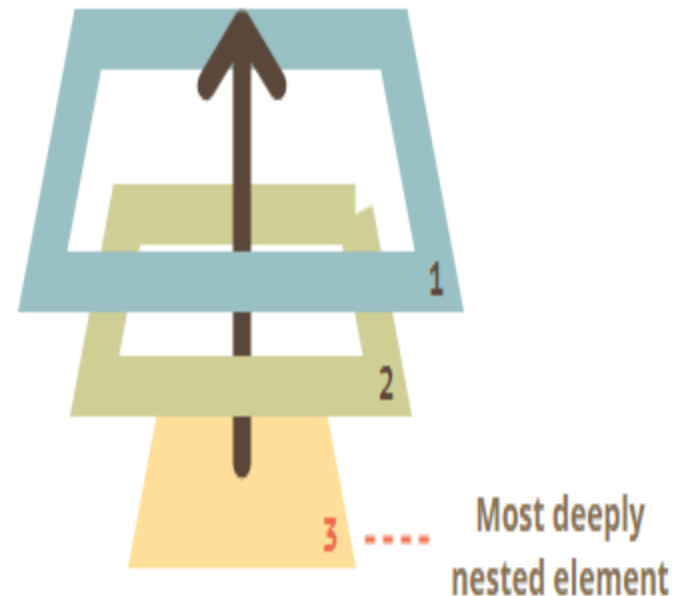
**p → div → form.**

- The process is called “bubbling”, because events “bubble” from the inner element up through parents like a bubble in the water.



# EVENT BUBBLING

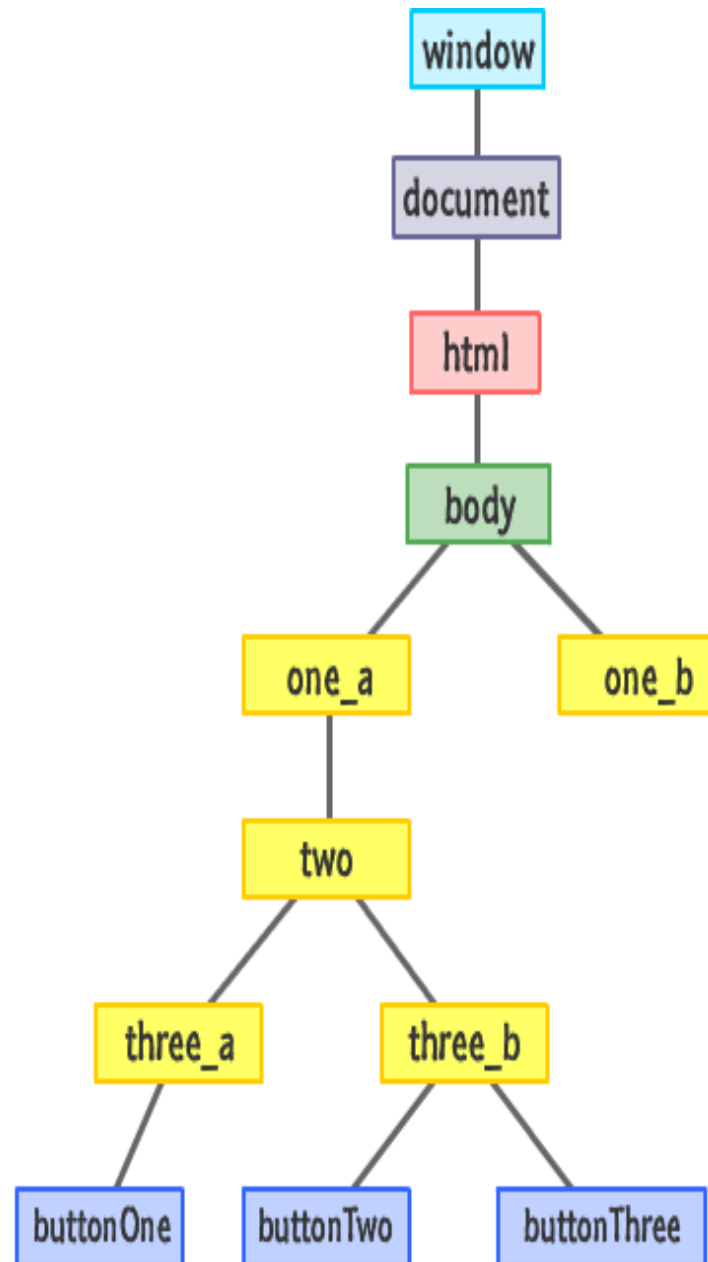
- The deepest element which triggered the event is called the target or, the originating element.
- When handlers trigger on parents:
  - `event.target/srcElement` - remains the same originating element.
  - `this` - is the current element, the one event has bubbled to, the one which runs the handler.

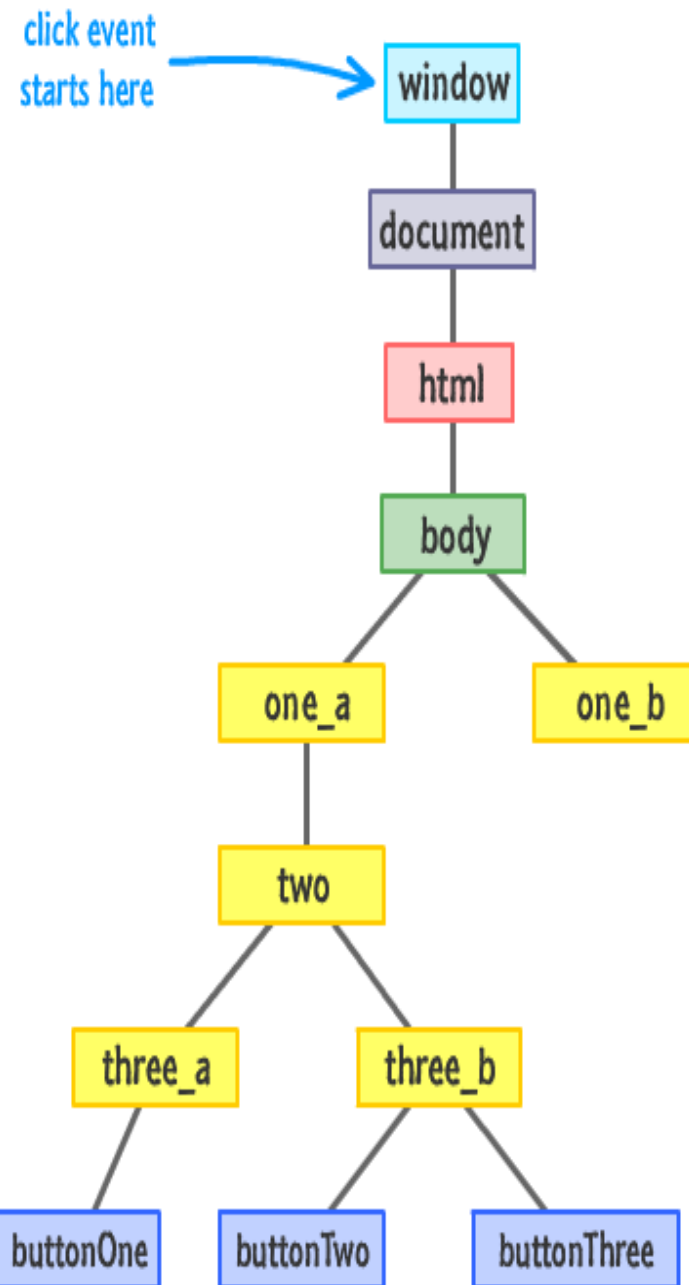


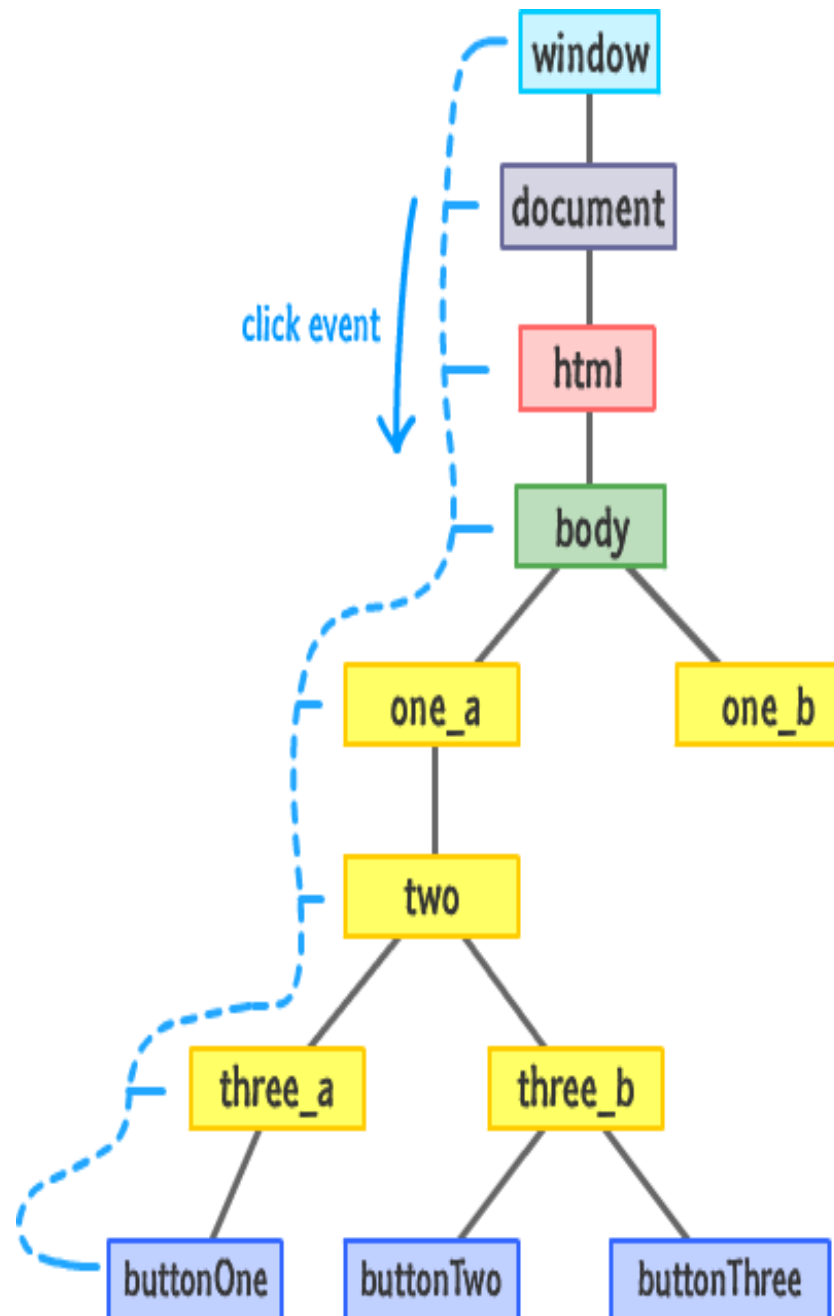
# EVENT BUBBLING

```
<body id="theBody" class="item">
 <div id="one_a" class="item">
 <div id="two" class="item">
 <div id="three_a" class="item">
 <button id="buttonOne" class="item">one</button>
 </div>
 <div id="three_b" class="item">
 <button id="buttonTwo" class="item">two</button>
 <button id="buttonThree" class="item">three</button>
 </div>
 </div>
 <div id="one_b" class="item">
 </div>
</body>
```

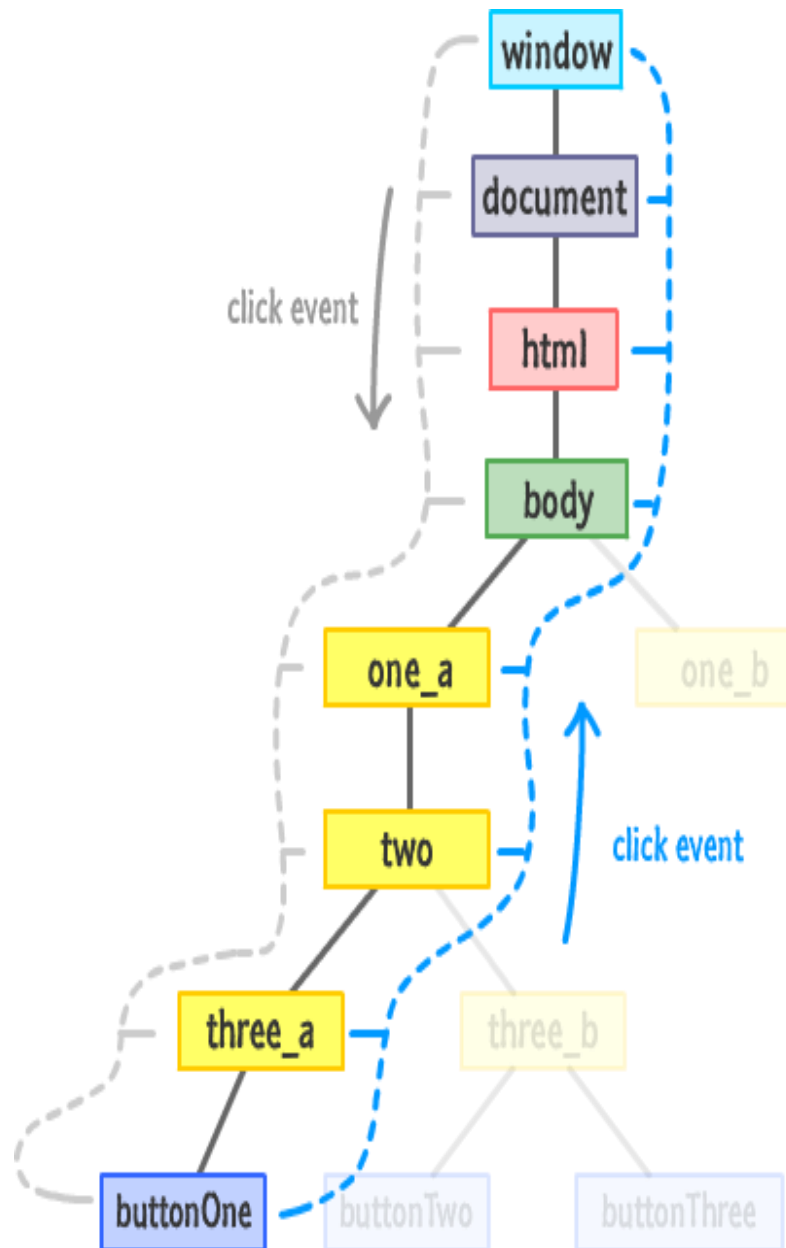


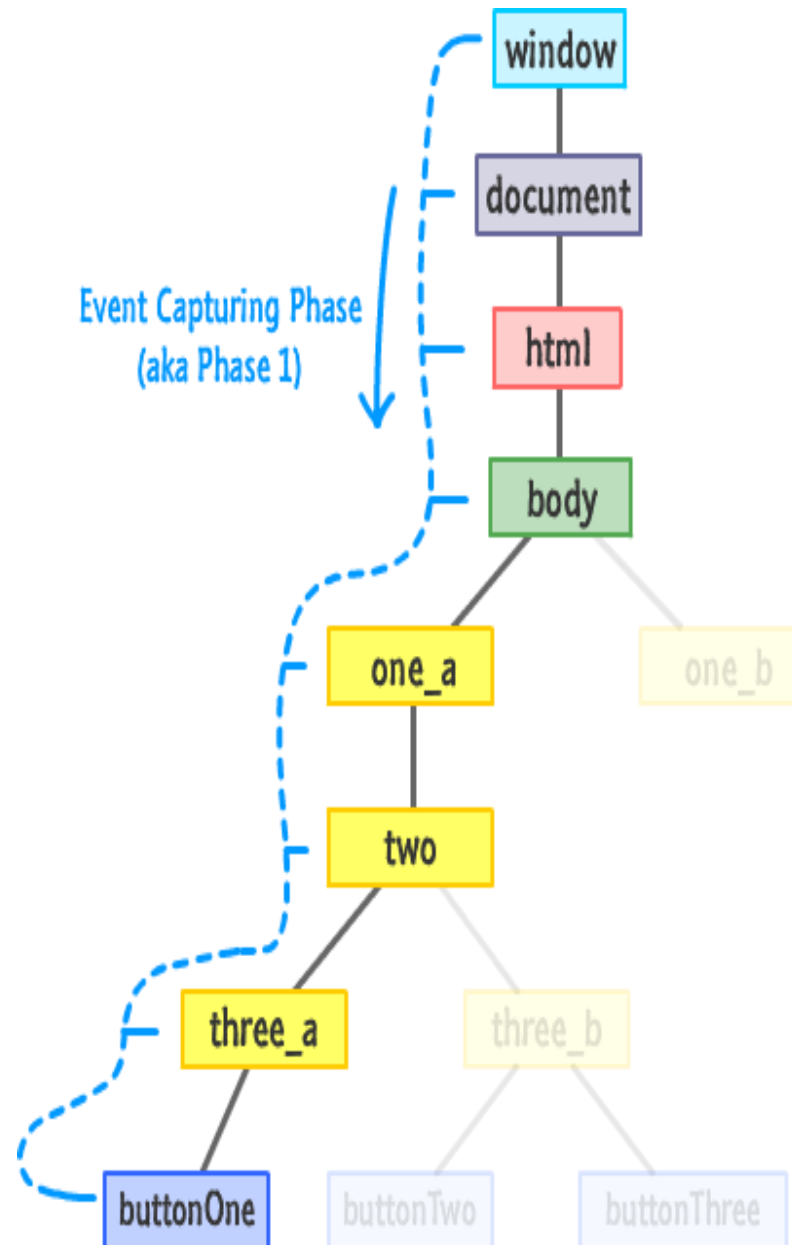


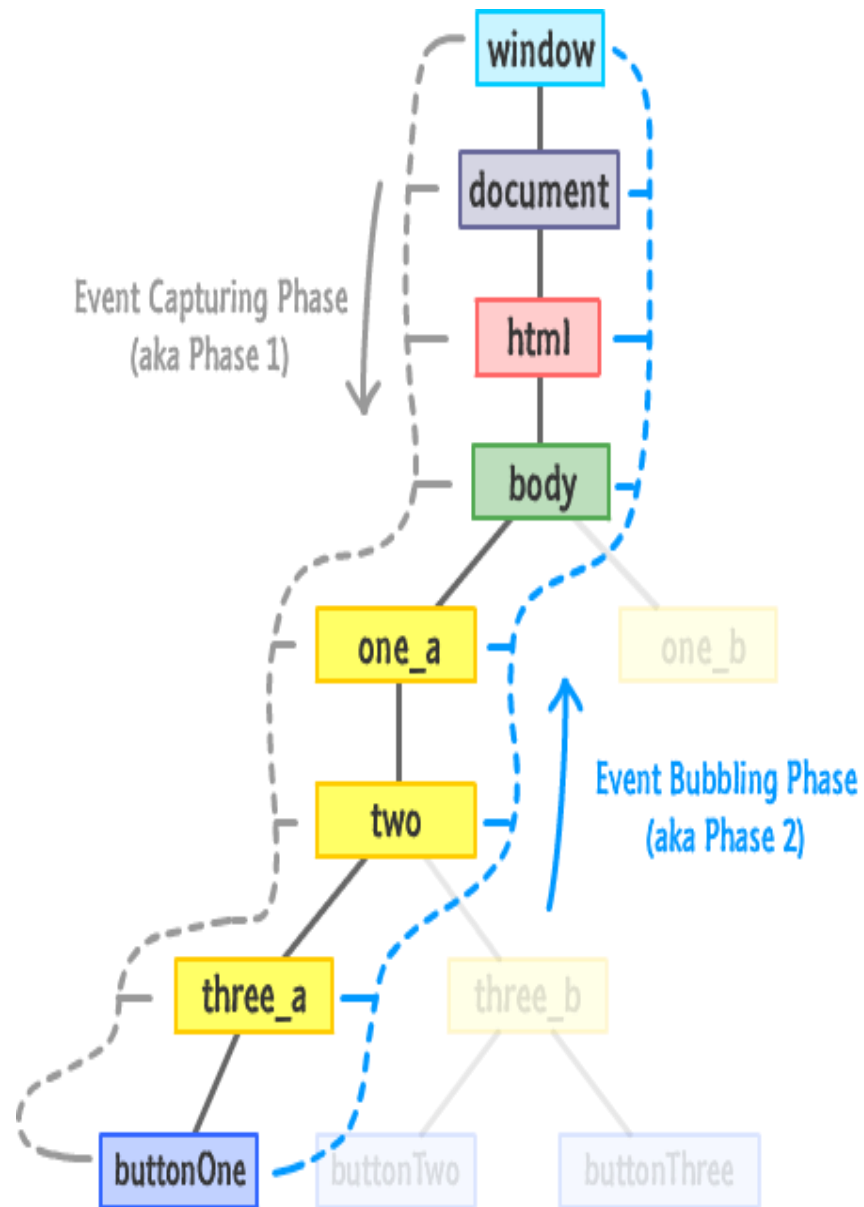






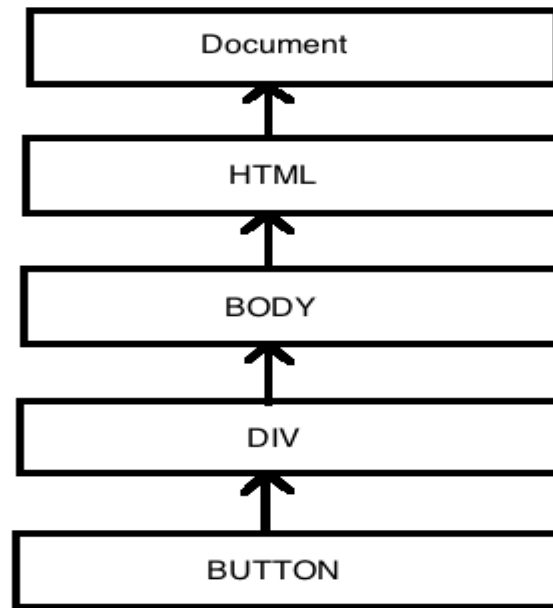






# EVENT BUBBLING

- Event Bubbling is the event starts from the deepest element or target element to its parents, then all its ancestors which are on the way to bottom to top.
- At present, all the modern browsers have event bubbling as the default way of event flow.



Event Bubbling



# EVENT BUBBLING

- If we choose not to specify the third argument for the phase altogether,
- When you don't specify the third argument, the default behavior is to listen to your event during the bubbling phase.
- It's equivalent to passing in a false value as the argument.

```
item.addEventListener("click",dosomething);
```



# STOPPING BUBBLING

- 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.
- But any handler may decide that the event has been fully processed and stop the bubbling.
- The method for it is **event.stopPropagation()**.
- We create a nested menu. Each submenu handles clicks on its elements and calls stopPropagation so that the outer menu won't trigger.



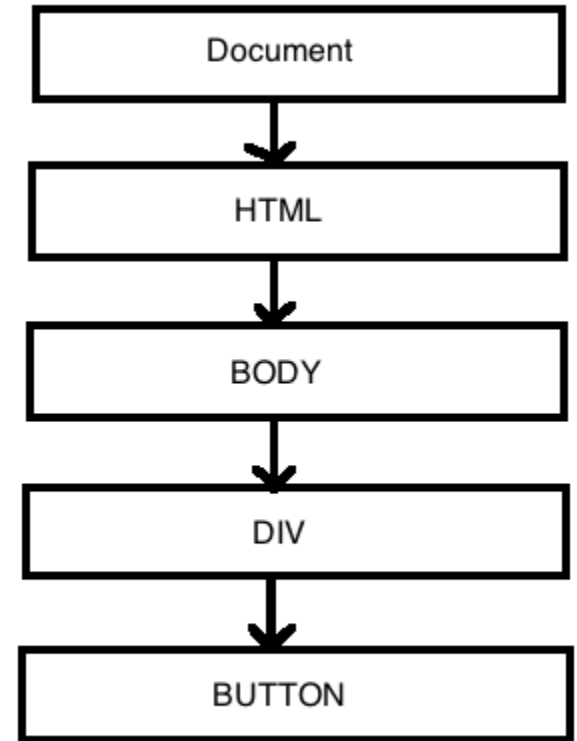
# EVENT CAPTURING

- *Event capturing* is the opposite of bubbling (events are handled at higher levels first, then *sink* down to individual elements at lower levels).
- **Event Capturing** is the event starts from top element to target element.
- Modern browser doesn't support event capturing by default but we can achieve that by code in JavaScript.
- Event capturing is supported in fewer browsers and rarely used; notably, Internet Explorer prior to version 9.0 does not support event capturing.



# EVENT CAPTURING

- When we click on the button first run the function which is attached on div , after that onclick function of button runs.
- This is due to Event Capturing.
- First run the event which is attached with parent element then event target.



Event Capturing





# EVENT BUBBLING & CAPTURING

- An argument of
  - True- means that you want to listen to the event during the capture phase.
  - False- this means you want to listen for the event during the bubbling phase.
- To listen to an event across both the capturing and bubbling phases, you can simply do the following:

```
item.addEventListener("click",dosomething,True);
item.addEventListener("click",dosomething,false);
```

