Mouse and Keyboard Events

Mouse events: mouseover, mouseout, mouseleave, mousemove, mouseenter, mousedown, mouseup.

Keyboard events: keyup, keydown, keypress.

The **mouseDown()** function is triggered when the mouse button is pressed down over this paragraph, and sets the color of the text to red.

The **mouseUp()** function is triggered when the mouse button is released, and sets the color of the text to green.

MouseEnter: The function bigImg() is triggered when the user moves the mouse pointer onto the image.

MouseLeave: The function normalImg() is triggered when the mouse pointer is moved out of the image.

MouseMove and Out: When the mouse is moved over the div, the p element will display the horizontal and vertical coordinates of your mouse pointer, whose values are returned from the clientX and clientY properties on the MouseEvent object.

KeyUp: A function is triggered when the user releases a key in the input field.

KeyDown: A function is triggered when the user is pressing a key in the input field.

KeyPress: A function is triggered when the user is pressing a key in the input field.

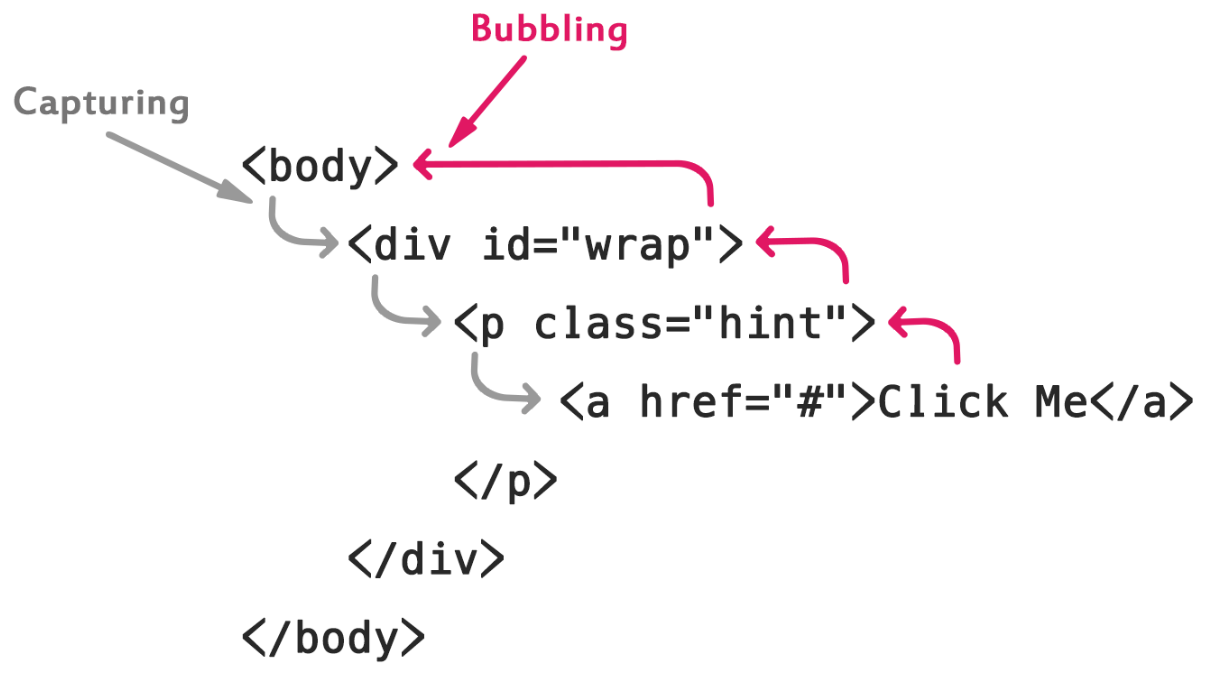
Propagation of Event

Event propagation is a mechanism that defines how events propagate or travel through the DOM tree to arrive at its target and what happens to it afterward.

Let's understand this with the help of an example, suppose you have assigned a click event handler on a hyperlink (i.e. <a> element) which is nested inside a paragraph (i.e. <p> element). Now if you click on that link, the handler will be executed.

But, instead of link, if you assign the click event handler to the paragraph containing the link, then even in this case, clicking the link will still trigger the handler. That's because events don't just affect the target element that generated the event—they travel up and down through the DOM tree to reach their target. This is known as event propagation

In modern browser event propagation proceeds in two phases: capturing, and bubbling phase. Before we proceed further, take a look at the following illustration:



```
<body>
    <div id="wrap">
        <p class="hint">
            <a href="#">Click Me</a>
        </p>
    </div>
</body>
```

The concept of event propagation was introduced to deal with the situations in which multiple elements in the DOM hierarchy with a parent-child relationship have event handlers for the same event, such as a mouse click. Now, the question is which element's click event will be handled first when the user clicks on the inner element—the click event of the outer element, or the inner element.

The Capturing Phase

In the capturing phase, events propagate from the Window down through the DOM tree to the target node. For example, if the user clicks a hyperlink, that click event would pass through the <html> element, the <body> element, and the <p> element containing the link.

Also if any ancestor (i.e. parent, grandparent, etc.) of the target element and the target itself has a specially registered capturing event listener for that type of event, those listeners are executed during this phase. Let's check out the following example:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Event Capturing Demo</title>
<style type="text/css">
    div, p, a{
        padding: 15px 30px;
        display: block;
        border: 2px solid #000;
        background: #fff;
    }
</style>
</head>
<body>
<div id="wrap">DIV
    <p class="hint">P
        <a href="#">A</a>
    </p>
</div>
```

```
<script>
    function showTagName() {
        alert("Capturing: "+ this.tagName);
    }

    var elems = document.querySelectorAll("div, p, a");
    for(let elem of elems) {
        elem.addEventListener("click", showTagName, true);
    }
</script>
</body>
</html>
```

Note: Also, event capturing only works with event handlers registered with the addEventListener() method when the third argument is set to true. The traditional method of assigning event handlers, like using onclick, onmouseover, etc. won't work here. Please check out the JavaScript event listeners chapter to learn more about event listeners.

The Bubbling Phase

In the bubbling phase, the exact opposite occurs. In this phase event propagates or bubbles back up the DOM tree, from the target element up to the Window, visiting all of the ancestors of the target element one by one. For example, if the user clicks a hyperlink, that click event would pass through the <p> element containing the link, the <body> element, the <html> element, and the document node.

Also, if any ancestor of the target element and the target itself has event handlers assigned for that type of event, those handlers are executed during this phase. In modern browsers, all event handlers are registered in the bubbling phase, by default. Let's check out an example:

```html
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Event Bubbling Demo</title>
<style type="text/css">
    div, p, a{
        padding: 15px 30px;
        display: block;
        border: 2px solid #000;
        background: #fff;
    }
</style>
</head>
<body>
<div onclick="alert('Bubbling: ' + this.tagName)">DIV
    <p onclick="alert('Bubbling: ' + this.tagName)">P
        <a href="#" onclick="alert('Bubbling: ' + this.tagName)">A</a>
    </p>
</div>
</body>
</html>
```

Note: Event bubbling is supported in all browsers, and it works for all handlers, regardless of how they are registered e.g. using onclick or addEventListener() (unless they are registered as capturing event listener). That's why the term event propagation is often used as a synonym of event bubbling.

Accessing the Target Element

The target element is the DOM node that has generated the event. For example, if the user clicks a hyperlink, the target element is the hyperlink.

The target element is accessible as event.target, it doesn't change through the event propagation phases. Additionally, the this keyword represents the current element (i.e. the element that has a currently running handler attached to it). Let's check out an example:

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Event Target Demo</title>
<style type="text/css">
    div, p, a{
        padding: 15px 30px;

        display: block;
        border: 2px solid #000;
        background: #fff;
    }
</style>
</head>
<body>
<div id="wrap">DIV
    <p class="hint">P
        <a href="#">A</a>
    </p>
</div>
```

```
<script>
    // Selecting the div element
    var div = document.getElementById("wrap");

    // Attaching an onclick event handler
    div.onclick = function(event) {
        event.target.style.backgroundColor = "lightblue";

        // Let the browser finish rendering of background color
before showing alert
        setTimeout(() => {
            alert("target = " + event.target.tagName + ", this = " +
this.tagName);
            event.target.style.backgroundColor = ''
        }, 0);
    }
</script>
</body>
</html>
```

Note: The fat arrow (=>) sign we've used in the example above is an arrow function expression. It has a shorter syntax than a function expression, and it would make the this keyword behave properly. Please check out the tutorial on ES6 features to learn more about arrow function.

Stopping the Event Propagation

You can also stop event propagation in the middle if you want to prevent any ancestor element's event handlers from being notified about the event.

For example, suppose you have nested elements and each element has onclick event handler that displays an alert dialog box. Normally, when you click on the inner element all handlers will be executed at once, since event bubble up to the DOM tree.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Event Propagation Demo</title>
<style type="text/css">
    div, p, a{
        padding: 15px 30px;
        display: block;
        border: 2px solid #000;
        background: #fff;
    }
</style>
</head>
<body>
<div id="wrap">DIV
    <p class="hint">P
        <a href="#">A</a>
    </p>
</div>
```

```
<script>
    function showAlert() {
        alert("You clicked: "+ this.tagName);
    }

    var elems = document.querySelectorAll("div, p, a");
    for(let elem of elems) {
        elem.addEventListener("click", showAlert);
    }
</script>
</body>
</html>
```

To prevent this situation you can stop event from bubbling up the DOM tree using the event.stopPropagation() method. In the following example click event listener on the parent elements will not execute if you click on the child elements.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Stop Event Propagation Demo</title>
<style type="text/css">
    div, p, a{
        padding: 15px 30px;
        display: block;
        border: 2px solid #000;
        background: #fff;
    }
</style>
</head>
<body>
<div id="wrap">DIV
    <p class="hint">P
        <a href="#">A</a>
    </p>
</div>
```

```
<script>
    function showAlert(event) {
        alert("You clicked: "+ this.tagName);
        event.stopPropagation();
    }

    var elems = document.querySelectorAll("div, p, a");
    for(let elem of elems) {
        elem.addEventListener("click", showAlert);
    }
</script>
</body>
</html>
```