JavaScript Absolute Beginner's Guide Part 4

Part 4: Dealing with Events

Chapter 28: Events

- An event is nothing more than a signal. It communicates that something has just happened.
  - This something could be a mouse click.
  - It could be a key press on your keyboard.
  - It could be your window getting resized.
  - It could just be your document simply getting loaded.
    - The thing to take away is that your signal could be one of hundreds of somethings that are built in to the JavaScript DOM API...or custom somethings that you created just for your app alone.
- To work with events, there are two things you need to do:
  1. Listen for events
  2. React to events

Listening for Events

- Almost everything you do inside an application results in an event getting fired.
- The thing to note is that your application is bombarded by events constantly whether you intended to have them get fired or not.
  - Our task is to tell your application to listen only to the events we care about.
- `addEventListener` is responsible for being eternally vigilant so that it can notify another part of your application when an interesting event gets fired.
  - The way you use this function looks as follows:

```
source.addEventListener(eventName, eventHandler, useCapture);
```

    - **The Source**: You call `addEventListener` via an element or object that you want to listen for events on. Typically, that will be a DOM element, but it can also be your document, window, or any other object that just happens to fire events.
    - **The Event Name**: The first argument you specify to the `addEventListener` function is the name of the event you are interested in listening to.
    - **The Event Handler**: The second argument requires you to specify a function that will get called when the event gets overheard.

        - **To Caputre of Not to Capture**: The last argument is made up of either a `true` or a `false`.
            - `true` means you want to listen for the event during the capture phase (phase 1).
            - `false` means you want to listen for the event during the bubbling phase (phase 2).
    - Real world example:

```
document.addEventListener("click", changeColor, false);
```

        - Our `addEventListener` in this example is attached to the `document` object. When a `click` event is overheard, it calls the `changeColor` function (aka the event handler) to react to the event.

Reacting to Events

- The only distinction between a typical function and one that is designated as the event handler is that your event handler function is specifically called out by name in an `addEventListener` call (and receives an Event object as its argument):

```
document.addEventListener("click", changeColor, false);
function changeColor() {
    // I am important!!!
}
```

    - Note that when calling `changeColor` in the `addEventListener` function, we do not use parentheses at the end like `changeColor()`

The Event Arguments and the Event Type

- Your event handler does more than just get called when an event gets overheard by an event listener.
    - It also provides access to the underlying event object as part of its arguments.
- To access this event object easily, we need to modify your event handler signature to support this argument.
- Here is an example:

```
function myEventHandler(e) {
    // event handlery stuff
}
```

    - At this point, your event handler is still a plain ol' boring function. It just happens to be a function that takes one argument...the event argument!

        - You can go with any valid identifier for the argument, but I tend to go with `e` because that is what a lot of people do.
    - The event argument points to an `Event` object, and this object is passed in as part of the event firing.
        - This `Event` object contains properties that are relevant to the event that was fired.
        - An event triggered by a mouse click will have different properties when compared to an event triggered by your keyboard key press, a page load, an animation, and a whole lot more.
- Despite the variety of events and resulting event objects you can get, there are certain properties that are common. This commonality is made possible because all event objects are derived from a base `Event` type (technically, an `Interface`).
- Some of the popular properties from the `Event` type that you will use are as follows:
    - `currentTarget`
    - `target`
    - `preventDefault`
    - `stopPropagation`
    - `type`

Removing an Event Listener

- Sometimes, you will need to remove an event listener from an element.
- The way you do that is by using the `removeEventListener` function:

```
something.removeEventListener(eventName, eventHandler, useCapture);
```

    - As you can see, this function takes the exact number and type of arguments as an `addEventListener` function.
    - The reason for that is simple:
        - When you are listening for an event on an element or object, JavaScript uses an `addEventListener`'s `eventName`, `eventHandler`, and the `true` or `false` value to uniquely identify that event listener. To remove this event listener, you need to specify the exact same arguments.
- Here is an example:

```
document.addEventListener("click", changeColor, false);
document.removeEventListener("click", changeColor, false);
```

- Resources:
    - https://www.w3schools.com/JSREF/obj_event.asp

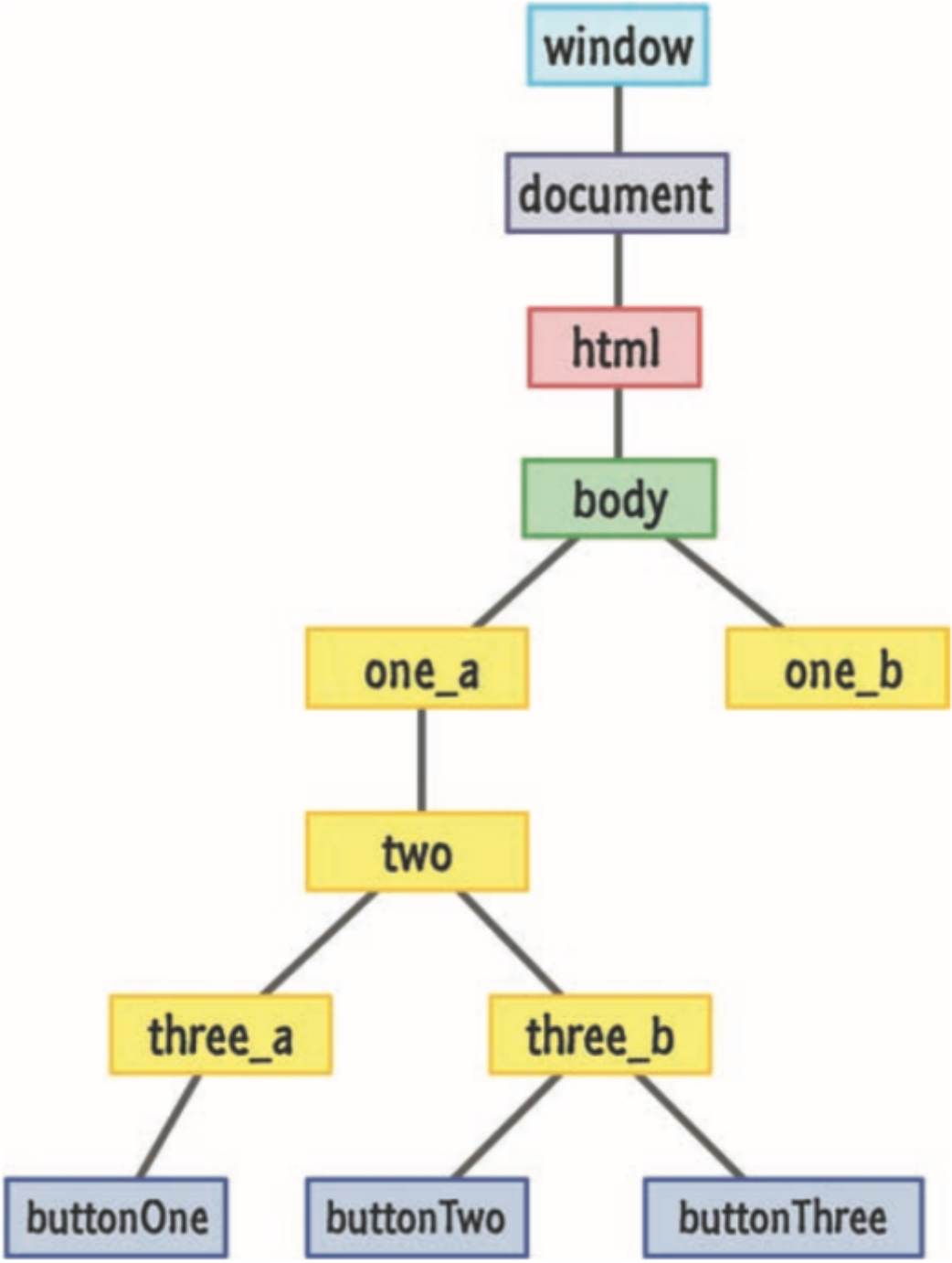
        - The Event Object
    - https://www.w3schools.com/JSREF/dom_obj_event.asp
        - List of Events
    - https://www.w3.org/TR/DOM-Level-3-Events/#events-module
        - Another list of Events
    - https://developer.mozilla.org/en-US/docs/Web/Events
        - List of Events in Firefox
    - https://developer.mozilla.org/en-US/docs/Web/API/Event
        - The Event Object in Firefox

#Chapter 29: Event Bubbling and Capturing

- To better help us understand events and their lifestyle, let's frame all of this in the context of a simple example. Here is some HTML we'll refer to:

```
<!DOCTYPE html>
<html>

<head>
    <title>Events!</title>
</head>

<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>
    <div id="one_b" class="item">
```

```
    </div>
    <script>

    </script>
</body>

</html>
```



Event Goes Down. Event Goes Up.

- Let's say that we click event that is fired on the `buttonOne` element.

- Your click event (just like almost every other JavaScript event) does not actually originate at the element that you interacted with.
- Instead it starts at the root of your document, in our case at `window`, then from the root, the event makes its way through the pathways of the DOM and stops at the element that triggered the event (aka the **event target**), which in our case is `buttonOne`.
  - The initial pathway our event would take is:

```
window
document
html
body
one_a
two
three_a
buttonOne
```

  - This means that if you were to listen for a click event on `body`, `one_a`, `two`, or `three_a`, the associated event handler will get fired.
- Now, once your event reaches its target, it doesn't stop, the event keeps going by retracing its steps and returning back to the root.
  - The return pathway our event would take is:

```
buttonOne
three_a
two
one_a
body
html
document
window
```

Meet the Phases

- The initial pathway is called Phase 1, or the **Event Capturing Phase**
- The return pathway is called Phase 2, or the **Event Bubbling Phase**
- Now recall the `addEventListener` function:

```
item.addEventListener("click", doSomething, true);
item.addEventListener("click", doSomething, false);
```

  - This third argument specifies whether you want to listen for this event during the capture phase.

    - An argument of `true` means that you want to listen to the event during the capture phase.
    - If you specify `false`, this means you want to listen for the event during the bubbling phase.
- **NOTE:** If you do not specify `true` or `false` for the third argument, the behavior defaults to the bubbling phase, aka it defaults to `false`

Who Cares?

- Normally, it won't make a difference whether `true` or `false` is specified. However there are some select scenarios where it will matter.
- A few of these scenarios are as follows:
    1. Dragging an element around the screen and ensuring the drag still happens even if the element I am dragging slips out from under the cursor
    2. Nested menus that reveal submenus when you hover over them
    3. You have multiple event handlers on both phases, and you want to focus only on the capturing or bubbling phase event handlers exclusively
    4. A third party component/control library has its own eventing logic and you want to circumvent it for your own custom behavior
    5. You want to override some built-in/default browser behavior such as when you click on the scrollbar or give focus to a text field

Event, Interrupted

- A full life of an event starts and ends at the root of the document.
    - Sometimes it is desirable to prevent the event from reaching a section of the document, thus cutting its life short.
- To end the life of an event, you have the `stopPropagation` method on your `Event` object:

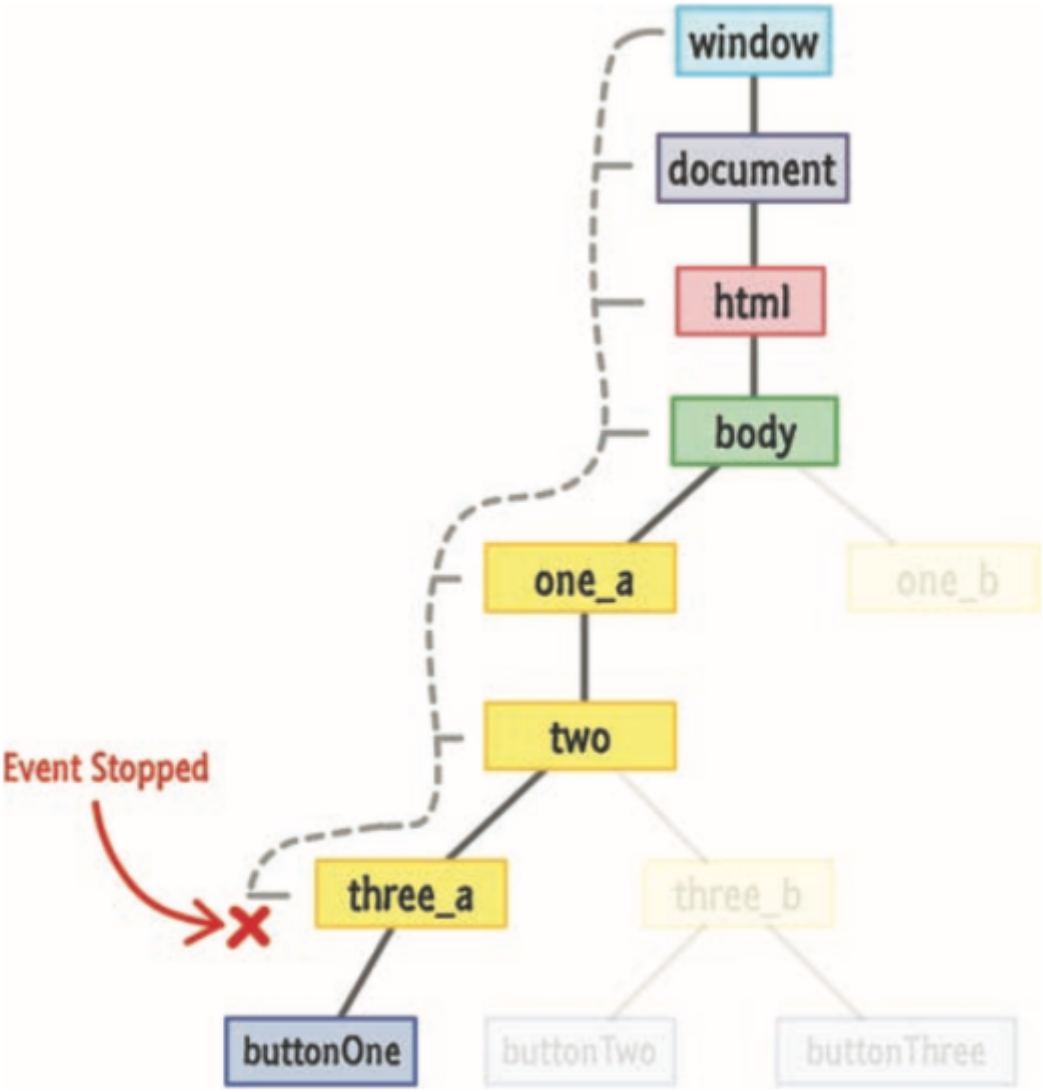
```
function handleClick(e) {
    e.stopPropagation();
    // do something
}
```

    - The `stopPropagation` method prevents your event from continuing through the phases.
- Example: let's say that you are listening for the click event on the three_a element and wish to stop the event from propagating. The code for preventing the propagation will look as follows:

```
var theElement = document.querySelector("#three_a");
theElement.addEventListener("click", doSomething, true);

function doSomething(e) {
```

```
    e.stopPropagation();
}
```

- When you click on `buttonOne`, here is what our event's path will look like:



- Your click event will steadfastly start moving down the DOM tree and notifying every element on the path to `buttonOne`.
  - Because the `three_a` element is listening for the click event during the capture phase, the event handler associated with it will get called: `function doSomething(e){ e.stopPropagation();}`
- In general, events will not continue to propagate until an event handler that gets activated is fully dealt with.
  - Because `three_a` has an event listener specified to react on a click event, the `doSomething` event handler gets called. Your event is in a holding pattern at this point until the `doSomething` event handler executes and returns.
- The click event will never reach the `buttonOne` element nor get a chance to bubble back up. So tragically sad.

- Another function you may encounter is `preventDefault` which will stop the default behavior from occurring, which is dependent upon the object which is calling it.
    - In other words, this function needs to be called when reacting to an event on the element whose default reaction you want to ignore.
    - Example:

```
function overrideScrollBehavior(e) {
    e.preventDefault();
    // do something
}
```

Chapter 30: Mouse Events

- Most popular mouse events:
    - `click`
    - `dblclick`
    - `mouseover`
    - `mouseout`
    - `mouseenter`
    - `mouseleave`
    - `mousedown`
    - `mouseup`
    - `mousemove`
    - `contextmenu`
    - `mousewheelandDOMMouseScroll`

Clicking Once and Clicking Twice

- This `click` event is fired when you click on an element.
    - More specifically, the `click` event is fired when you use your mouse to press down on an element and then release the press while still over that same element.
- `click` example:

```
var button = document.querySelector("#myButton");
button.addEventListener("click", doSomething, false);

function doSomething(e) {
```

```
    console.log("Mouse clicked on something!");
}
```

- The `dblclick` event is fired when you quickly repeat a click action a double number of times, and the code for using it looks as follows:

```
var button = document.querySelector("#myButton");
button.addEventListener("dblclick", doSomething, false);

function doSomething(e) {
    console.log("Mouse clicked on something...twice!");
}
```

The Very Click-like Mousing Down and Mousing Up Events

- Two events that are almost subcomponents of the `click` event are the `mousedown` and `mouseup` ones.
- The `mousedown` event is fired when the mouse is pressed and held.
- The `mouseup` event is fired when the mouse is released after a click happens.
- **NOTE:** If the element you pressed down on and released from are the same element, the `click` event will also fire.
- You can see all of this from the following snippet:

```
var button = document.querySelector("#myButton");
button.addEventListener("mousedown", mousePressed, false);
button.addEventListener("mouseup", mouseReleased, false);
button.addEventListener("click", mouseClicked, false);

function mousePressed(e) {
    console.log("Mouse is down!");
}
function mouseReleased(e) {
    console.log("Mouse is up!");
}
function mouseClicked(e) {
    console.log("Mouse is clicked!");
}
```

Mousing Over and Mousing Out

- The classic hover over and hover out scenarios are handled by the appropriately titled `mouseover` and `mouseout` events respectively.

- The `mouseover` event fires when the mouse enters a specified area.
- The `mouseout` event fires when the mouse leaves a specified area.
- Here is a snippet of these two events in action:

```
var button = document.querySelector("#myButton");
button.addEventListener("mouseover", hovered, false);
button.addEventListener("mouseout", hoveredOut, false);

function hovered(e) {
    console.log("Hovered!");
}
function hoveredOut(e) {
    console.log("Hovered Away!");
}
```

`mousemove`

- This event fires a whole lotta times as your mouse moves over the element you are listening for the `mousemove` event on.
- An example of the `mousemove` event in code:

```
var button = document.querySelector("#myButton");
button.addEventListener("mousemove", mouseIsMoving, false);

function mouseIsMoving(e) {
    console.log("Mouse is on the run!");
}
```

The Context Menu

- The `contextmenu` event is fired just before the right-click menu appears.
    - The main reason this funciton exists is so that this menu is prevented from appearing when you right-click or use a context menu keyboard button or shortcut.
- Here is an example of how you can prevent the default behavior where the context menu appears:

```
document.addEventListener("contextmenu", hideMenu, false);

function hideMenu(e) {
    e.preventDefault();
}
```

The `MouseEvent` Properties

The Global Mouse Position

- The `screenX` and `screenY` properties return the distance your mouse cursor is from the top-left location of your primary monitor (top-left corner = (0,0) in x, y coordinates).
- Here is a very simple example of the screenX and screenY properties at work:

```
document.addEventListener("mousemove", mouseMoving, false);

function mouseMoving(e) {
    console.log(e.screenX + " " + e.screenY);
}
```

The Mouse Position Inside the Browser

- The `clientX` and `clientY` properties return the x and y position of the mouse relative to your browser's (technically, the browser viewport's) top-left corner.
- Example code:

```
var button = document.querySelector("#myButton");
button.addEventListener("mousemove", mouseMoving, false);

function mouseMoving(e) {
    console.log(e.clientX + " " + e.clientY);
}
```

Detecting Which Mouse Button Was Clicked

- To figure out which mouse button was pressed, you have the `button` property:
    - This property returns a 0 if the left mouse button was pressed,
    - a 1 if the middle button was pressed,
    - and a 2 if the right mouse button was pressed.
- The code for using the button property to check for which button was pressed looks exactly as you would expect:

```
document.addEventListener("mousedown", buttonPress, false);

function buttonPress(e) {
    if (e.button == 0) {
        console.log("Left mouse button pressed!");
    } else if (e.button == 1) {
        console.log("Middle mouse button pressed!");
    } else if (e.button == 2) {
        console.log("Right mouse button pressed!");
```

```
    } else {
        console.log("Things be crazy up in here!!!");
    }
}
```

- In addition to the `button` property, you also have the `buttons` and `which` properties that sorta do similar things to help you figure out which button was pressed. Google them if you want to know more.

Dealing with the Mouse Wheel

- You have the `mousewheel` event that is used by Internet Explorer and Chrome and the `DOMMouseScroll` event used by Firefox.
- The way you listen for these mouse wheel-related events is just the usual:

```
document.addEventListener("mousewheel", mouseWheeling, false);
document.addEventListener("DOMMouseScroll", mouseWheeling, false);
```

    - The `mousewheel` and `DOMMouseScroll` events will fire the moment you scroll the mouse wheel in any direction.
- The event arguments for a `mousewheel` event contain a property known as `wheelDelta`.
- For the `DOMMouseScroll` event, you have the `detail` property on the event argument.
    - Both of these properties are similar in that their values change from positive or negative depending on what direction you scroll the mouse wheel.
    - The thing to note is that they are inconsistent in what sign they go with.
        - The `wheelDelta` property associated with the `mousewheel` event is positive when you scroll up on the mouse wheel. It is negative when you scroll down.
        - The exact opposite holds true for `DOMMouseScroll`'s `detail` property. This property is negative when you scroll up, and it is positive when you scroll down.
- Handling this wheelDelta and detail inconsistency is pretty simple...as you can see in the following snippet:

```
function mouseWheeling(e) {
    var scrollDirection = e.wheelDelta || -1 * e.detail;
    if (scrollDirection > 0) {
        console.log("Scrolling up! " + scrollDirection);
    } else {
        console.log("Scrolling down! " + scrollDirection);
    }
}
```

- The `scrollDirection` variable stores the value contained by the `wheelData` property or the `detail` property.

Chapter 31: Keyboard Events

- 3 main keyboard events:
  - `keydown`
    - fired when you press down on a key on your keyboard.
  - `keyup`
    - fired when you release a key that you just pressed
  - `keypress`
    - `keypress` event is fired only when you press down on a key that displays a character (letter, number, and the like).
      - 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.
      - If you press and release a key that doesn't display anything on the screen (such as the spacebar, arrow key, or function keys), all you will see are the `keydown` and `keyup` events fired.
- Here is an example of listening to the three keyboard events on the `window` object:

```
window.addEventListener("keydown", dealWithKeyboard, false);
window.addEventListener("keypress", dealWithKeyboard, false);
window.addEventListener("keyup", dealWithKeyboard, false);

function dealWithKeyboard(e) {
    // gets called when any of the keyboard events are overheard
}
```

The Keyboard Event Properties

- Let's revisit our `dealWithKeyboard` event handler that you saw earlier. In that event handler, the keyboard event is represented
  by the `e` argument that is passed in:

```
function dealWithKeyboard(e) {
    // gets called when any of the keyboard events are overheard
}
```

- This argument contains a handful of properties:

- `KeyCode`
  - Every key you 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.
- `ctrlKey`, `altKey`, `shiftKey`
  - These three properties return a true if the Ctrl key, Alt key, or Shift key are pressed.
- `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.

- **CAUTION:** The `charCode` and `keyCode` properties are currently marked as deprecated by the web standards people at the W3C. It's replacement might be the mostlyunsupported `code` property.

Checking That a Particular Key Was Pressed

- The following example shows how to use the `keyCode` property to check if a particular key was pressed:

```
window.addEventListener("keydown", checkKeyPressed, false);

function checkKeyPressed(e) {
    if (e.keyCode == "65") {
        console.log("The 'a' key is pressed.");
    }
}
```

- If you wanted to check the `charCode` and use the keypress event, here is what the above example would look like:

```
window.addEventListener("keypress", checkKeyPressed, false);

function checkKeyPressed(e) {
    if (e.charCode == 97) {
        alert("The 'a' key is pressed.");
    }
}
```

Doing Something When the Arrow Keys Are Pressed

- You see this most often in games where pressing the arrow keys does something interesting. The following snippet of code shows how that is done:

```
window.addEventListener("keydown", moveSomething, false);

function moveSomething(e) {
    switch(e.keyCode) {
        case 37:
            // left key pressed
            break;
        case 38:
            // up key pressed
            break;
        case 39:
            // right key pressed
            break;
        case 40:
            // down key pressed
            break;
    }
}
```

Detecting Multiple Key Presses

- An interesting case revolves around detecting when you need to react to multiple key presses.
- What follows is an example of how to do that:

```
window.addEventListener("keydown", keysPressed, false);
window.addEventListener("keyup", keysReleased, false);
var keys = [];
function keysPressed(e) {
    // store an entry for every key pressed
    keys[e.keyCode] = true;

    // Ctrl + Shift + 5
    if (keys[17] && keys[16] && keys[53]) {
        // do something
    }

    // Ctrl + f
    if (keys[17] && keys[70]) {
```

```
        // do something
        // prevent default browser behavior
        e.preventDefault();
    }
}

function keysReleased(e) {
    // mark keys that were released
    keys[e.keyCode] = false;
}
```

Chapter 32: Page Load Events and Other Stuff

The Things That Happen During Page Load

- When you load a webpage code runs on the page between the time that you sent the request and viewing your fully loaded page.
- When exactly the code runs depends on a combination of the following things that all come alive at some point while your page is getting loaded:
  - The `DOMContentLoaded` event
  - The `load` Event
  - The `async` attribute for script elements
  - The `defer` attribute for script elements
  - The location your scripts live in the DOM
- 3 main stages of loading a webpage:
  1. The first stage is when your browser is about to start loading a new page.
     - A request has been made to load a page, but nothing has been downloaded yet.
  2. With the second stage where the raw markup and DOM of your page has been loaded and parsed.
     - The thing to note about this stage is that external resources like images and linked stylesheets have not been parsed. You only see the raw content specified by your page/document's markup.
  3. The final stage is where your page is fully loaded with any images, stylesheets, scripts, and other external resources making their way into what you see.
     - This is the stage where your browser's loading indicators stop animating, and this is also the stage you almost always find yourself in when interacting with your HTML document.

The `DOMContentLoaded` and `load` Events

- The `DOMContentLoaded` event fires at the end of Stage #2 when your page's DOM is fully parsed.
- The `load` event fires at the end of Stage #3 once your page has fully loaded.
- Below is a snippet of these events in action:

```
document.addEventListener("DOMContentLoaded", theDomHasLoaded, false);
window.addEventListener("load", pageFullyLoaded, false);

function theDomHasLoaded(e) {
    // do something
}
function pageFullyLoaded(e) {
    // do something again
}
```

    - You use these events just like you would any other event, but the main thing to note about these events is that you need to listen to them from the `document` element
        - You can technically listen to these events on other elements, but for page loading scenarios, you want to stick with the `document` element.

- `DOMContentLoaded` and `load` are important because if you have any code that relies on working with the DOM such as anything that uses the `querySelector` or `querySelectorAll` functions, you want to ensure your code runs only after your DOM has been fully loaded. If you try to access your DOM before it has fully loaded, you may get incomplete results or no results at all.
- A sure-fire way to ensure you never get into a situation where your code runs before your DOM is ready is to listen for the `DOMContentLoaded` event and let all of the code that relies on the DOM to run only after that event is overheard:

```
document.addEventListener("DOMContentLoaded", theDomHasLoaded, false);
function theDomHasLoaded(e) {
    var images = document.querySelectorAll("img");
    // do something with the images
}
```

- For cases where you want your code to run only after your page has fully loaded, use the `load` event.
    - *Author's Note: "I never had too much use for the `load` event at the document level outside of checking the final dimensions of a loaded image or creating a crude progress bar to indicate progress."*

Scripts and Their Location in the DOM

- Your browser parses your DOM sequentially from the top to the bottom. Any script elements that are found along the way will get parsed in the order they appear in the DOM.

- Below is a very simple example where you have many script elements:

```
<!DOCTYPE html>
<html>
<body>
    <h1>Example</h1>
    <script>
        console.log("inline 1");
    </script>
    <script src="external1.js"></script>
    <script>
        console.log("inline 2");
    </script>
    <script src="external2.js"></script>
    <script>
        console.log("inline 3");
    </script>
</body>
</html>
```

  - It doesn't matter if the script contains inline code or references something external. All scripts are treated the same and run in the order in which they appear in your document.
  - Using the above example, the order in which the scripts will run is as follows: inline 1, external 1, inline 2, external 2, and inline 3.

- **Because your DOM gets parsed from top to bottom, your script element has access to all of the DOM elements that were already parsed. Your script has no access to any DOM elements that have not yet been parsed.**
  - By putting your script element at the bottom of your page as shown earlier, the end behavior is identical to what you would get if you had code that explicitly listened to the `DOMContentLoaded` event.
    - Additionally, page processing freezes when a script element is being parsed so if you have a long script your page will appear incomplete or frozen to the user if you have a script at the top of the page without the use of `DOMContentLoaded`
  - If you really want to have your script elements at the top of your DOM, ensure that all of the code that relies on the DOM runs after the `DOMContentLoaded` event gets fired.

Script Elements—Async and Defer

`async`

- The `async` attribute allows a script to run asynchronously:

```
<script async src="someRandomScript.js"></script>
```

- If a script element is being parsed, it could block your browser from being responsive and usable.
  - By setting the `async` attribute on your script element, you avoid that problem altogether.
  - Your script will run whenever it is able to, but it won't block the rest of your browser from doing its thing.
  - BUT you must realize that your scripts marked as async will not always run in order... The only guarantee you have is that your scripts marked with `async` will start running at some mysterious point before the `load` event gets fired.

`defer`

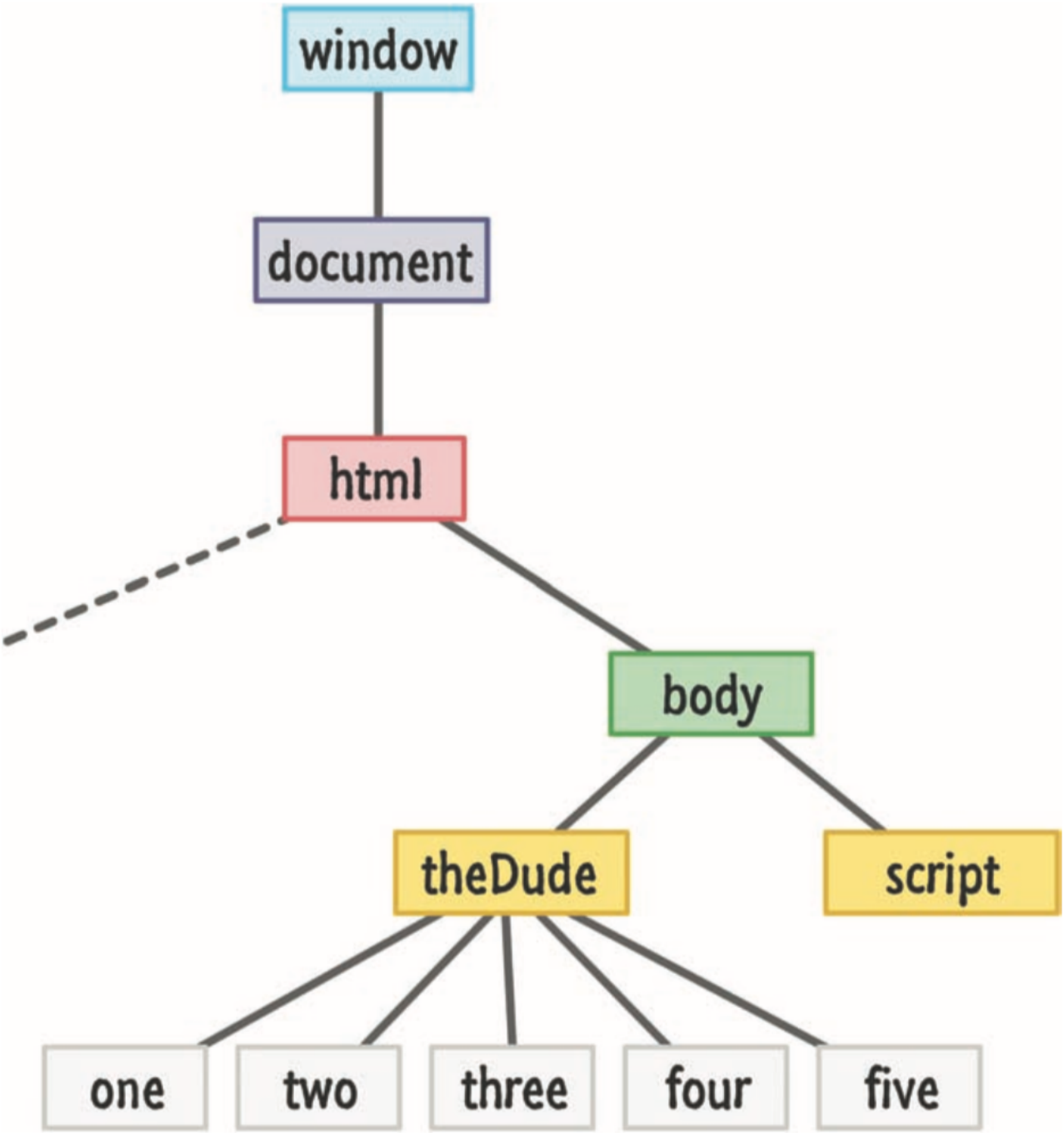
- The defer attribute is a bit different from async:

```
<script defer src="someRandomScript.js"></script>
```

  - Scripts marked with `defer` run in the order in which they were defined, but they only get executed at the end just a few moments before the `DOMContentLoaded` event gets fired.
- Additional Resources:
  - https://requirejs.org/docs/start.html
    - RequireJS

Chapter 33: Handling Events For Multiple Elements

- When listening to multiple events your goal should be to minimize the amount of event listeners you use, because each event listenter takes up a small bit of memory and it adds up. Additionally, it is always a good practice to try and minimize code, particularly duplicate code.
- Imagine we have a case where you want to listen for the click event on any of the sibling elements whose `id` values are `one`, `two`, `three`, `four`, and `five`. Let's complete our imagination by picturing the DOM as follows:



- At the very bottom, we have the elements we want to listen for events on. They all share a common parent with an element whose `id` value is `theDude`.

A Terrible Solution

- A terrible solution would be to add 5 event listeners, 1 for each of the 5 buttons. That would look like this:

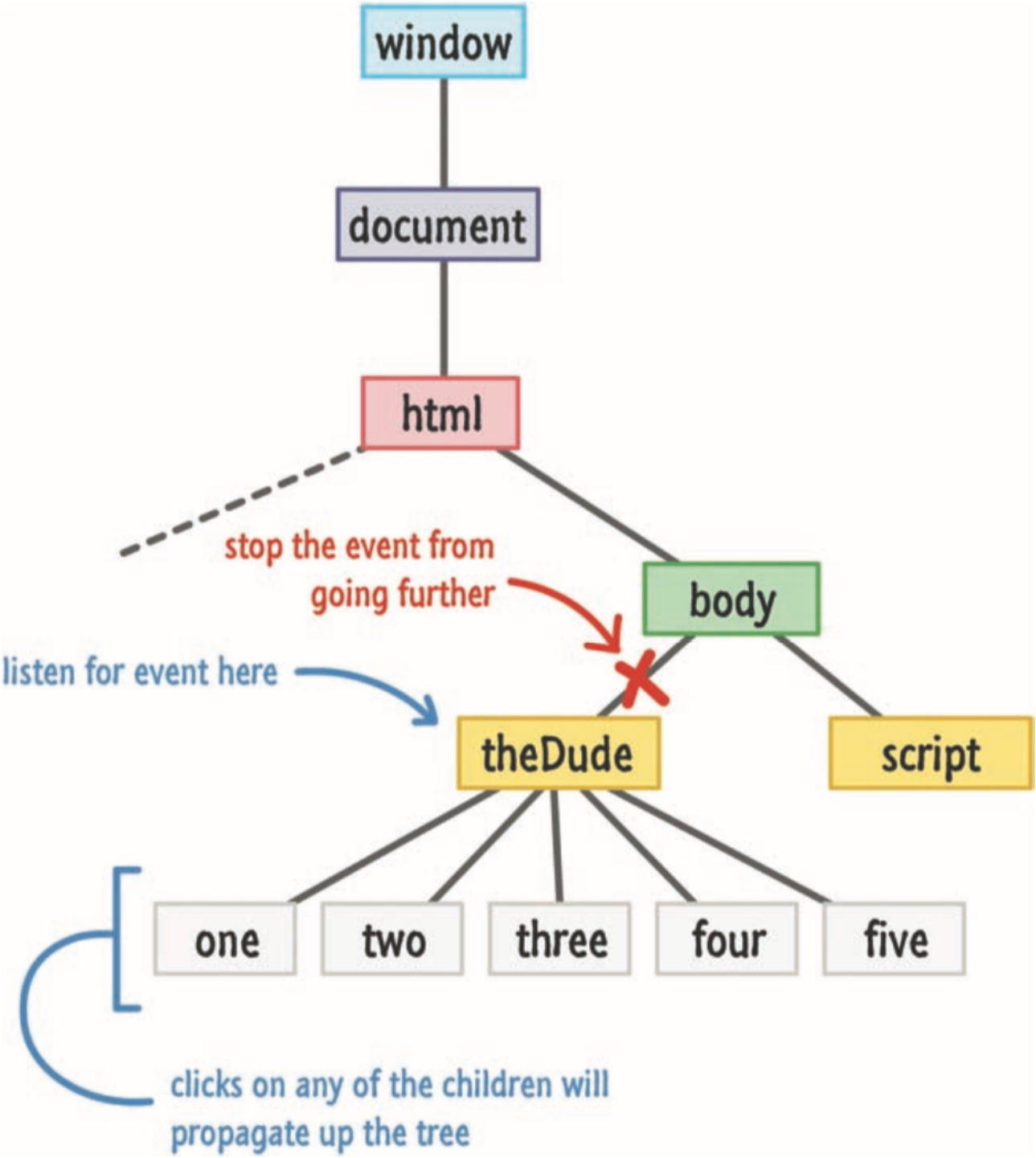
```
var oneElement = document.querySelector("#one");
var twoElement = document.querySelector("#two");
var threeElement = document.querySelector("#three");
var fourElement = document.querySelector("#four");
var fiveElement = document.querySelector("#five");
oneElement.addEventListener("click", doSomething, false);
```

```
twoElement.addEventListener("click", doSomething, false);
threeElement.addEventListener("click", doSomething, false);
fourElement.addEventListener("click", doSomething, false);
fiveElement.addEventListener("click", doSomething, false);

function doSomething(e) {
    var clickedItem = e.target.id; alert("Hello " + clickedItem);
}
```

    - Obviously, **this is a terrible solution** due to the duplicate code and excessive use of event listeners via the `addEventListener` function.

A Good Solution

- In a good solution for this problem we would just use 1 event listener. This is possible because all of our elements we want to listen to share a common parent.
- The solution steps are as follows:
    1. Create a single event listener on the parent `theDude` element.
    2. When any of the `one`, `two`, `three`, `four`, or `five` elements are clicked, rely on the propagation behavior that events possess and intercept them when they hit the parent `theDude` element.
    3. (Optional) Stop the event propagation at the parent element just to avoid having to deal with the event obnoxiously running up and down the DOM tree.



- The code that translates what the diagram and the three steps represent is as follows:

```
var theParent = document.querySelector("#theDude");
theParent.addEventListener("click", doSomething, false);

function doSomething(e) {
    if (e.target !== e.currentTarget) {
        var clickedItem = e.target.id;
        alert("Hello " + clickedItem);
    }
    e.stopPropagation();
}
```

- We listen for the event on the parent `theDude` element.
- There is only one event listener to handle this event called `doSomething`.
- This event listener will get called each time `theDude` element is clicked along with any children that get clicked as well.
    - We only care about click events relating to the children, and the proper way to ignore clicks on this parent element is to simply avoid running any code if the element the click is from (aka the event target) is the same as the event listener target (aka `theDude` element).
        - The target of the event is represented by `e.target`, and the target element the event listener is attached to is represented by `e.currentTarget`.
        - By simply checking that these values not be equal, you can ensure that the event handler doesn't react to events fired from the parent element that you don't care about.
- To stop the event's propagation, we simply call the `stopPropagation` method.

END OF BOOK