COMP 1073
Client-Side Scripting

Lesson 5 – Part 1

Interacting with the User Through Events

Georgian

Objectives

In this Lesson we will learn about:

1. **Attaching an Event**
2. **Mouse and Keyboard Events**
3. **Touch and Orientation Events**

Interacting with the User Through Events

Introduction – Interacting with the User Through Events

- **Events** are triggered in the browser when something happens; users do not always initiate them, but they often do.
- For example, "load" is an event that **fires** when a page is first loaded; although the user did technically do something to make the page load, it's more indirect than, for instance, a user clicking a link.
- **Click** is another event, which happens when a user clicks something.
- These events need to be attached to **DOM** elements to be executed.
- The statement "click a link" means that the event "click" is being attached to a "link," and then a **function** is executed.
- Whether that **function** is designed to open a URL from an **`href`** attribute, to **`hide / show`** content, or maybe even load someone's contact information from a beautifully designed **JSON** object from a previous chapter, the same model applies:
  1. Get a DOM node.
  2. Attach an event to it.
  3. Execute a specific **function** when the event is triggered.

Interacting with the User Through Events (Cont'd)

- You may hear this model referred to as **event-driven JavaScript**.
- Just like some **back-end technologies** are best at a class-heavy object-oriented model, JavaScript is best at dealing with an **event-driven model**.
- This means that you write code, something happens (an event), and then it gets executed (a function / method).
- There are constant debates over the best way to **format** JavaScript: whether to let it go wild, whether to use a back-end **model-view-controller (MVC)** method, how object-oriented you should get, or whether to use an event-driven model.
- Some technologies are strong in a certain area and are meant for a very object-oriented model.

Attaching an Event

Attaching an Event

- Now that you are thinking about events, we can talk about attaching them to **DOM elements**.
- Events and functions are great; they create human-readable statements like “alert a warning message on click.”
- That’s all well and good, but by itself that statement means very little; JavaScript needs more information to complete the task at hand.
- You need to tell the browser exactly **what to do** when something is clicked.
- So statements like “alert a warning message on click of the button” or “validate this form when it is submitted” are a lot more meaningful to a program.

Attaching an Event (Cont'd)

- Attaching an event to a DOM element is one of the most important places to keep the methodologies behind progressive enhancement in mind; it's very easy to create events that (for example) communicate to the server or a data set and overlook the fact that there is no server-side fallback being created in case JavaScript is turned off.
- As previously mentioned, coding this way will not only ensure that your content is available to all users, but it will also cut down on the amount of JavaScript you'll need to write.
- This will result in better performance and a smoother user experience.

Attaching an Event (Cont'd)

- There are a lot of ways to attach an event to a DOM element.
- Some are just flat out wrong (using inline JavaScript in your HTML), some are less wrong, and some are much better.
- We'll be going over two methods:
  - Event handlers
  - Event listeners

Event Handlers

- **Event handlers** are what I refer to as the “less wrong” way of attaching an event to a DOM element.
- That may have been a little dramatic—there’s nothing technically wrong with using an event handler.
- The method just has some **drawbacks**, which are fixed by using other methods.
- **Event handlers** are very simple to use because they’re very human-readable.

Event Handlers (Cont'd)

- In the following Listing you can see that we're using a **`button`** element with an **`id`** value of " btn " and attaching a click event to it in the form of "**`onclick`**"; then we set that equal to an **anonymous function**, which will alert the text "clicked the button."

```
/* wrap it in an anonymous function to contain the variables */

(function(){

    // HTML: <button type="button" id="btn">a button</button>

    // save the button to a variable
    var btn = document.getElementById("btn");

    // Use an event handler to attach the "onclick" event
    btn.onclick = function(){

        // alert inside this anonymous callback function
        alert('clicked the button');

    }

})();
```

Event Handlers (Cont'd)

- When using event handlers, the normal event you would attach is always prefaced with the word "**on**".
- That's why we're using "**onclick**" rather than just "**click.**"
- This pattern is applied to all events: **`onsubmit`**, **`onchange`**, **`onfocus`**, **`onblur`**, **`onmouseover`**, **`onmouseout`**, **`ontouchstart`**, **`ongesturestart`**, and so on.
- You get the point; every **event** is **`on<event>`** when using **event handlers**.
- For the most part, **event handlers** are fine; they have great browser support and actually predate the **DOM standard**.
- The one issue with handlers is that you can attach only a single function to a specific event of a DOM node.

Event Listeners

- **Event listeners** have a similar function to that of event handlers in that you still need a **DOM element** to attach them to, you still need to **identify the event**, and you still need a **function to call**.
- The difference is that you don't have the limitation of the handler when assigning multiple functions to the same DOM element and event.
- It unties your hands a little as a developer, and I personally prefer working with them a little more than I do handlers.
- The **syntax** of an event listener is a little different from a handler—most notably, the **lack of an equal sign** and the fact that it looks more like a native method.

Event Listeners (Cont'd)

- The following Listing shows an example of an event listener using an **anonymous function**.

```
// event listener with an anonymous function
element.addeventListenter("event", function(){

    // stuff you want the function to do

}, false);
```

- An event listener is made up of four parts:
  - The DOM element ("element" in the above Listing)
  - The **event** type ("event" in the above Listing)
  - The **`eventListener`**, aka the function (**anonymous function** in the above Listing)
  - A **`Boolean`** value used to initiate something called "**capture**" (set to **`false`** in the above Listing).

Event Listeners (Cont'd)

- Other than **initiating capture**, most arguments in the `addEventListener()` method are pretty straightforward, so let's go over capture very briefly.
- You will be setting this capture option to `false` 99% of the time.
- Setting it to `true` is like getting on a megaphone and announcing to all the parent DOM nodes that an event is firing on a particular node.
- Setting to `false` prevents this behavior because, for the most part, it is unnecessary.
- It's officially called "event propagation."

Event Listeners (Cont'd)

- In the following Listing you will see a more real-world example of a basic event listener being set on a button, which has an ID value of " btn ."
- We're using the **`getElementById()`** method to save it to a variable and add the event listener to it.
- The event is defined as "click ," and we're executing an **anonymous function** to alert a message and setting the capture Boolean to **`false`**.

```
/* wrap it in an anonymous function to contain the variables */

(function(){

    /* HTML: <button type="button" id="btn">a button</button> */

    // save the button to a variable
    var btn = document.getElementById("btn");

    // attach a "click" listener to the button
    btn.addEventListener("click", function(){

        // alert inside this anonymous callback function
        alert('clicked the button');

    }, false);

})();
```

Browser Support

- **Support** is a small caveat of using the **`addEventListener()`** method.
- It's one of the instances where "it works where you expect it to work." Internet Explorer 8 and earlier do not support **`addEventListener()`**, but there is something that functions in almost the same way.
- The method used in **IE8** and earlier is called **`attachEvent()`**; the most obvious difference between the two methods is that **`attachEvent()`** takes only two arguments.
- It excludes the **capture Boolean** value because before Internet Explorer 9, there was no event propagation model available in IE.
- So it makes sense why the value didn't exist.
- Because there is **no overlap** in support of the two methods, it either exists or it doesn't; you don't need to detect the browser version, you can detect for the presence of **`addEventListener()`** and modify your script with an **`if`** statement accordingly.
- If the method exists, use it; if not, use **`attachEvent()`.**

Browser Support (Cont'd)

- The following Listing shows how to use an **`if` statement** to detect for the presence of **`addEventListener`** and conditionally let the browser decide which method to use.

```
var btn = document.getElementById("btn");

if(btn.addEventListener){

    /* if eventListener is supported do this */
    btn.addEventListener("click", function(){

        // alert inside this anonymous callback function
        alert("clicked the button");

    }, false);
} else {
    /* if it's not supported do this (for IE8 and below) */
    element.attachEvent("click", function(){

        // alert inside this anonymous callback function
        alert("Clicked the button");

    });
}
```

Binding Events

- When **binding** (attaching) events to DOM elements, there are a few weird things to note.
- Remember that when a **function** is **nested** inside an object, it's actually called a **method**?
- Well, something similar is going on with event listeners and event binding.
- When a function is **inside** an **`addEventListener()`** method, it's called an **event listener**.
- It's still a function when it's outside the **`addEventListener()`** method, but when it's in the context of that particular method it's referred to as an event listener.
- It's the listener you're adding to the event.

Binding Events (Cont'd)

- There is one other strange thing about calling functions inside the **`addEventListener()`** method.
- They don't have parentheses like they've had up to this point in the book.
- Because the parentheses mean "call this now" and we don't want to do that in a listener, you leave them off.

Binding Events (Cont'd)

- The following Listing shows how to use **a predefined function** as a listener by dropping the parentheses off.
- If the parentheses were to be left on, the function would **execute immediately** on load of the page, even if it's inside the **`addEventListener()`** method.

```
// save the DOM element you want to attach an event to
var btn = document.getElementByID("btn");

// define your function normally
function alertMessage(){
    alert("clicked the button");
}

// or use a predefined function (event handler), "alertMessage"
btn.addEventListener("click", alertMessage, false);
```

Binding Events (Cont'd)

- In previous lessons, we discussed adding a dynamic nature to functions by **passing arguments** through them.
- With no parentheses, you can't pass any arguments into the function.
- In order to pass arguments into a function while using **`addEventListener()`**, you need to use the function as a **callback** instead by nesting it inside an **anonymous function**.

Binding Events (Cont'd)

- This may sound a little confusing, but the following Listing shows you how to use an **anonymous function** and a **callback** function to achieve this goal.

```
// save the DOM element you want to attach an event to
var btn = document.getElementByID("btn");

// define your function normally
function alertMessage(message) {
    alert(message);
}

// or use a predefined function (event handler), "alertMessage"
btn.addEventListener("click", function() {

    // callback function!
    alertMessage("clicked the button");

}, false);
```

Unbinding Events

- Just like how you want to **bind** (attach) events to DOM elements, sometimes you want to **unbind** (detach) events from DOM nodes.
- Most of the time you'll probably **attach** an event and leave it alone, but if you want to **clean up after yourself**, you can certainly **remove** the listener just as easily as you can add it.
- Internet Explorer version 8 and earlier have their own methods for removing events (detaching events).

Unbinding Events

- The following Listing shows an example of how to remove and detach an event similar to the way they are added.

```
if(btn.removeEventListener){

    // if removeEventListener is supported
    btn.removeEventListener("click", alertMessage, false);
```

```
} else {

    // if removeEventListener isn't supported (IE8 and below)
    btn.detachEvent("click", alertMessage);

}
```

Mouse and Keyboard Events

Mouse and Keyboard Events

- The most **common events**, other than **`load`**, that you will use are most likely mouse and **keyboard-based** events.
- What are mouse and keyboard-based events? Events such as
  - **`click`**
  - **`focus`**
  - **`blur`**
  - **`change`**
  - **`mouseover`** (hover part 1)
  - **`mouseout`** (hover part 2)
  - **`submit`** (form submit)
- There are a lot of events in JavaScript.
- Some you'll probably never use, like **`double-click`**.
- The previous list contains the popular **mouse** and **keyboard** events.

Mouse and Keyboard Events (Cont'd)

- To properly go through the following example of these events, you'll first need some HTML to work with.
- We'll be jumping back into the address book application that we've been building off of for this lesson.
- The HTML will consist of a **simple search form** to type in the name of a contact and have the contact show up on the page.
- The following Listing shows the basic HTML search form you'll use to build this functionality.
- Ideally, this form should be hooked up to a back-end technology (PHP, Python, Ruby, and the like) via the form action to process the user's request upon submitting the form so it is accessible without JavaScript.
- It functions in a similar manner to a normal **search form**, but instead of searching millions of websites this form searches through the **JSON object** of contacts you made.

Mouse and Keyboard Events (Cont'd)

```
<!doctype html>
<html>
<head>

    <meta charset="utf-8">
    <title>Address Book</title>
    <style>
        .active { background: #ddd; }
        .hovering { background: #eee; }
        form > div { padding: 10px; }
    </style>
</head>
<body>

<h1>Address Book</h1>

<!-- ideally you would have this hooked up to a PHP (any backend) processing page so
➥it works without JavaScript as well, but we won't get into that -->

<form action="" method="get" id="search-form">

    <div class="section">
        <label for="q">Search address book</label>
        <input type="search" id="q" name="q" required placeholder="type a name">
    </div><!--/.section-->

    <div class="button-group">
        <button type="submit" id="btn-search">search</button>
        <button type="button" id="get-all">get all contacts</button>
    </div><!--/.button-group-->

</form>

<div id="output"></div><!--/#output-->

<!-- Keeping JS at the bottom, because we rock at performance -->
<script src="js/addressbook.js"></script>

</body>
</html>
```

- You can see in the HTML that we've prepped the interface a little, so it's ready for our JavaScript **functionality** by adding an empty **`div`** with an **`id`** of "output" to accept our data, and there is also a **secondary button** to get all the contacts into the address book.

Mouse and Keyboard Events (Cont'd)

- The following Listing shows the **JSON object** created to hold contact information.
- It is the same object from previous lessons, but you should feel free to add some new people to freshen things up.

```
var contacts = {
    "addressBook" : [
        {
            "name": "hillisha",
            "email": "hill@example.com",
        },
        {
            "name": "paul",
            "email": "cleveland@example.com",
        },
        {
            "name": "vishaal",
            "email": "vish@example.com",
        },
        {
            "name": "mike",
            "email": "grady@example.com",
        },
        {
            "name": "jamie",
            "email": "dusted@example.com",
        }
    ]
};
```

Mouse and Keyboard Events (Cont'd)

`click`

- **`click`** is the most common of all the native events you will encounter.
- It can be applied to **any element**.
- You want to be sure that the element you're binding a click event to is indeed a **clickable element**.
- There are two ways to tell if an element if clickable.
  - The first is to turn off all CSS and JavaScript in the browser and click it; if something happens, you have a clickable element. If not, you don't.
  - The second is to **tab through** an interface and press **Enter / Return** when you get to that element. You will notice in your address book form that you should be able to tab through the interface and press your **Return / Enter** key to submit the form.
- A lot of elements are clickable.
- A heading (**h1**), for example, is technically clickable, but because you can't execute that click with anything but a mouse (you can't tab to it and press **Return / Enter**), you should not attach a click event to it no matter how easy it would be.

Mouse and Keyboard Events (Cont'd)

`click` (Cont'd)

- All the previous event examples in the lesson have been based on the **`click` event**, so I won't spend a lot of time rehashing the same information.
- We can jump right into the address book application you've been building.
- This functionality is going to be tied to the **Get All Contacts button**.
- You can probably guess what we're going to build.
- In the following Listing the first step is to define an object to hold all our methods (aka functions).
- We're going to call it "adr" and the first method will be "getAllContacts".
- Inside that method will be the **function loop** you defined in the previous lesson to parse through all the contacts in your **JSON object** and output them in the page.

Mouse and Keyboard Events (Cont'd)

```
/* define the adr object to hold your methods (aka functions) */
var adr = {

    getAllContacts : function(){

        /* save the DOM element we're putting the output in */
        var target = document.getElementById("output");

        /* save the contacts JSON object to a variable */
        var book = contacts.addressBook;

        /* save the length to a variable outside the loop for performance */
        var count = book.length;

        /* ready the loop! */
        var i;

        /* clear the contents of #output just in case there's something in there */
        target.innterHTML = "";

        /* as usual, check the count before looping */
        if(count > 0){
            /* loop through the contacts */
            for(i = 0; i < count; i = i + 1){

                var obj = book[i],

                target.innerHTML += '<p>' + obj.name + ', <a href="mailto:' + obj.
➥email + '">'+ obj.email +'</a><p>';

            } // end for loop
        } // end if count
    } // end method
} // end object
```

Mouse and Keyboard Events (Cont'd)

`focus` and `blur`

- Besides **`click`**, using **`focus`** and **`blur`** are the most common types of events when building a JavaScript application.
- The Web is made up of a bunch of **links** and **forms**, and **link** and **forms** are what work the best with the **`focus`** and **`blur`** events.
- A good example of a **`focus / blur`** action is activating a search form input field.
- Either by clicking in the field or using the **Tab key** to navigate to it would be a "focus," and whenever you deactivate the search box input field (such as clicking off of it), that is a **blur** event.

Mouse and Keyboard Events (Cont'd)

`focus` and `blur` (Cont'd)

- Think of **`focus`** and **`blur`** like **`click`** and **`unclick`** (if you could unclick something).
- They are **opposite behaviors** that mostly relate to **forms** and **form elements**, but they also can be applied to links.
- Because some elements work well with **`focus`** and **`blur`** and some don't, you first need to know whether an element is focusable before you attach this event to it.
- How do you tell if an element is focusable? Tab through a page; if you can reach the element you want, then it is **focusable**.
- If you're looking for a general rule, **link** and **form** input elements (text inputs, radios, check boxes, buttons) are all focusable elements.

Mouse and Keyboard Events (Cont'd)

`focus` and `blur` (Cont'd)

- In the **search form** you've been building we're going to use **`focus`** and **`blur`** to do something called **context highlighting**.
- **Context highlighting** is a method of bringing attention to a certain area of a form by changing **background color**.
- In this example, whenever a user focuses on the search field, we are going to add a class of "active" on the parent element (**`<div>`**), which we have already added CSS for to set a background color of gray.

Mouse and Keyboard Events (Cont'd)

`focus` and `blur` (Cont'd)

- The following Listing extends **`adr`** object to add two new methods that will be executed on **`focus`** and **`blur`**, respectively.
- After the methods are defined, you will see the **`focus`** and **`blur`** event listeners declared.

```
/* use the same adr method */
var adr = {

    /* ...previously defined methods go here... */

    // define the method (aka the function)
    addActiveSection : function() {
        // add a class of "active" to the wrapping div
        this.parentNode.setAttribute("class", "active");

    }, // end method, note the comma
    removeActiveSection: function() {

       // remove the class from the wrapping div
        this.parentNode.removeAttribute("class");

    }

} // end adr object

// save the search box to a variable
var searchField = document.getElementById(" q");

// activate the focus event on the search box
searchField.addEventListenter("focus", adr.addActiveSection, false);

// activate the blur event on the search box
searchField.addEventListenter("blur", adr.removeActiveSection, false);
```

Mouse and Keyboard Events (Cont'd)

`change`

- A **`change`** event is applied to **form elements** such as **select menus**, **radios buttons**, and **check boxes**.
- In radios and check boxes, a **change value** is triggered when the **box / `button`** is checked.
- This can happen in ways similar to how an element can get **focus**: clicking the **box / button directly**, click the **associated label**, and using the **Tab key** to navigate to it and pressing Enter/Return.
- In a **select menu**, the change event is triggered when a **new value** or option is selected by clicking or by keyboard navigation.
- Our **address book** doesn't have any use for the **change event**, but it is extremely useful to have in your toolkit, and is attached in the same way all the other events are attached—via **listeners**.

Mouse and Keyboard Events (Cont'd)

mouseover and mouseout (hovering)

- The `mouseover` event triggers when a user positions a cursor **over an element**.
- `mouseout` triggers in the opposite case, when the user removes the mouse cursor from the same element.
- The combination of these two events creates **a complete hover effect**.
- Let's go back to our address book application and create a simple event behavior of adding a class of "hovering" to the form on mouseover and removing that class on mouseout .
- You can see in the HTML snippet in the previous Listing that we have already reserved some CSS for this class.

Mouse and Keyboard Events (Cont'd)

mouseover and mouseout (hovering) (Cont'd)

- The following Listing shows how to add this **hover behavior** to the search form.
- This is an example of how to re-create a hover effect with **JavaScript** to add a class to our search form.
- This can be done with a small amount of CSS (**`form:hover{ /* css code */ }`**), and probably should be.
- Often, developers do use JavaScript where CSS would be a better option.

```
// save the element to a variable
var searchForm = document.getElementById("search-form");

/* use the same adr method */
var adr = {

    /* ...previously defined methods go here... */

    addHoverClass : function(){

        // add a class of "hovering" to the wrapping div
        searchForm.setAttribute("class", "hovering");

    }, // end method, note the comma
    removeHoverClass: function(){

        // remove all classes from the wrapping div
        searchForm.removeAttribute("class");

    } // end method

} // end object

// activate the focus event on the search box
searchForm.addEventListenter("mouseover", adr.addHoverClass, false);

// activate the blur event on the search box
searchForm.addEventListenter("mouseout", adr.removeHoverClass, false);
```

Mouse and Keyboard Events (Cont'd)

`submit`

- The **`submit`** event is triggered when a form is submitted, either by clicking a **Submit button**, tabbing to the Submit button and pressing **Enter / Return**, or sometimes just by pressing **Enter / Return** while a form element is in focus.
- Any way you can submit a form, this event gets triggered.
- In your **address book**, the submit event will be triggered to search for a name or string and return results.
- Normally, this search would be done on the server, but because the address book data is a **JSON object** inside the JavaScript file, we're going to want to use JavaScript to **parse** through the **JSON object**, search for the string entered in the search field, and return the results in the output area.

Mouse and Keyboard Events (Cont'd)

`submit` (Cont'd)

- The following Listing contains a **loop** of the data very similar to the **loop** you've been using all along, but instead of spitting out all the data, we're using the **`indexOf()`** method in JavaScript to search for a specific string.
- For example, if you were to type the letter "i" into the search box and submit the form, every contact that contains that letter should be returned.

Mouse and Keyboard Events (Cont'd)

`submit` (Cont'd)

- The **`indexOf()`** method will either return a match or the number **`-1`**, so all you have to do is check for **`-1`** and return the others.

```
            // loop through the contacts
            for(i = 0; i < count; i = i + 1) {

                // look through the name value to see if it contains the searchterm
➥string
                var obj = contacts.addressBook[i],
                    isItFound = obj.name.indexOf(searchValue);

                // anything other than -1 means we found a match
                if(isItFound !== -1) {
                    target.innerHTML += '<p>' + obj.name + ', <a href="mailto:' +
➥obj.email + '">'+ obj.email +'</a><p>';
                } // end if

            } // end for loop
```

Mouse and Keyboard Events (Cont'd)

Preventing Default Behavior

- **Preventing default behavior** is something you often need to do when executing JavaScript on DOM elements that have another behavior attached to them.
- For example, a link with an **href value** wants to go to another page, or in our case, a form that wants to submit somewhere to do a search query.
- Preventing default behavior is done inside the method.
- You first need to **pass the event** (form **submit** in our case) as an argument into the method.
- Then attach the `preventDefault()` method to it.

Mouse and Keyboard Events (Cont'd)

Preventing Default Behavior (Cont'd)

- This is depicted in the following Listing.

```
var adr = {

    search : function(event){

        event.preventDefault();

        /* continue the rest of the method here */

    }

}
```

Mouse and Keyboard Events (Cont'd)

keydown, keypress, and keyup

- **keydown**, **keypress**, and **keyup** refer to the user pressing the keys on the keyboard.
- **`keydown`** is the initial press, **`keyup`** is when the finger is lifted, and **`keypress`** is somewhere in the middle.
- **`keyup`** is the most popular of the three events.
- The **`keyup`** event tends to be **more accurate** in getting text input values, so it's more widely used.
- These events are often used to perform an **autocomplete** action on a search form.
- That is also how we will be using it on the address book application.

Mouse and Keyboard Events (Cont'd)

keydown, keypress, and keyup (Cont'd)

- Because an **autocomplete** action is nothing more than a basic search with fewer letters, the search method defined in the following Listing will be 100% reusable for this purpose.
- The only modification that needs to be made is at the **listener** level.
- Instead of using the **submit** event, you need to use a keyup event.
- The **event** trigger order of key events is as follows:
  - **keydown**
  - **keypress**
  - **Keyup**

Mouse and Keyboard Events (Cont'd)

keydown, keypress, and keyup (Cont'd)

- The following Listing shows how you would repurpose the search method in the previous Listing to function as an autocomplete function simply by changing the event trigger.
- This is one of the great advantages to event-driven JavaScript.

```
// activate autocomplete on keyup
searchField.addEventListener("keyup", addr.search, false);
```

Touch and Orientation Events

Touch and Orientation Events

- **Touch and orientation** events can be a little intimidating because they're relatively new additions to the language, but they are really new events that get attached the same way you've been attaching all the **non-touch** events like click and focus.
- In some cases, they can even use the same functions.
- Because the address book application is getting to a pretty stable point, for the **touch event** examples we're going to use a new, blank HTML file.

Touch and Orientation Events (Cont'd)

- The following Listing shows this blank file.
- Note that there is a **min-height** set on the **<body>**; because it's empty we want to give it some **height** so there is a surface available to touch.

```
<!doctype html>
<html lang="en">
<head>
    <title>Touch Events</title>
    <meta charset="utf-8">

    <style>
        body { min-height:600px;background:#ddd; }
    </style>

</head>
<body>

<h1>Touch Events Demo</h1>

<!-- JS at the bottom, because we still rock at performance -->
<script src="js/script.js"></script>

</body>
</html>
```

Touch and Orientation Events (Cont'd)

touchstart and touchend

- The paring of the **`touchstart`** and **`touchend`** are very common because they mark opposite events.
- The **`touchstart`** event is triggered when a user touches the screen, and the **`touchend`** event is triggered with the opposite action of untouching the screen.

Touch and Orientation Events (Cont'd)

```
/* Anonymous function wrapper again! */
(function(){

    var body = document.getElementsByTagName("body")[0];

    // declare an object to hold touch controls
    var touchControls = {

        pokeTheScreen : function(){

            // output a message to the body
            body.innerHTML += "you just poked me, how rude!<br>";

        }, stopPokingTheScreen: function(){

            // output another message to the body
             body.innerHTML += "please do not do that again.<br><br>";
        }

    } // end object

    // add event listeners to the body
    body.addEventListener("touchstart", touchControls.pokeTheScreen, false);
    body.addEventListener("touchend", touchControls.stopPokingTheScreen, false);

})();
```

Touch and Orientation Events (Cont'd)

`touchmove`

- The **`touchmove`** event is triggered when the user moves their finger on the screen.
- It is always preceded by the **`touchstart`** event.
- Naturally, you have to touch the screen before you can move your finger.
- This is often used to create **swipe gestures** or to move objects around the screen.

Touch and Orientation Events (Cont'd)

touchmove (Cont'd)

- In the following Listing we're creating a method inside the **touchControls** object that will output the text "moving!!" into the body while the **touchmove** event is being triggered.
- The event will trigger repeatedly until the movement stops.
- You can see this by moving your finger around the screen and observing the output (it should say "moving!!" a bunch of times).

```
/* Anonymous function wrapper again! */

(function(){

    // the same body variable, no need to redeclare it.
    var body = document.getElementsByTagName("body")[0];

    // declare an object to hold touch controls
    var touchControls = {

        /* previously defined methods here */

        showMovement : function(){

            // output a message to the body
            body.innerHTML += "moving!!<br>";

        } // end method
    } // end object

    // add event listeners to the body
    body.addEventListener("touchmove", touchControls.showMovement, false);

})();
```

Touch and Orientation Events (Cont'd)

`orientationchange`

- **`orientationchange`** is the only event related to touch that really isn't a touch event.
- As I mentioned earlier, this is the event that relies on the presence of an accelerometer in a device.
- An **accelerometer** is what allows the screen on your phone or tablet to rotate when you rotate the device.
- It also allows high-end gaming because it can return how many degrees the user is turning a device in one direction or another.
- There are **orientation settings** for **portrait**, **landscape**, **upside-down portrait**, and **upside-down landscape**, which can be returned with some work, but we're only interested in triggering the event that tells us that the device orientation has **changed**.

Touch and Orientation Events (Cont'd)

`Orientationchange` (Cont'd)

- In the following Listing you will see the method "**changedOrientation**" added to the **touchControls** object.
- This new method is set up to clear the contents of the body when the **orientationchange** event is triggered.
- If you add this method into the object, do some touching and dragging around; you should be able to clear the screen by rotating it to either portrait or landscape (depending on how you started off).

Touch and Orientation Events (Cont'd)

`Orientationchange` (Cont'd)

```
/* Anonymous function wrapper again! */

(function(){

    // the same body variable, no need to redeclare it.
    var body = document.getElementsByTagName("body")[0];

    // declare an object to hold touch controls
    var touchControls = {

        /* previously defined methods here */

        changedOrientation : function(){

            // clear out the body content
            body.innerHTML = "";

        } // end method
    } // end object

    // add event listeners to the body
    body.addEventListener("orientationchange", touchControls.changedOrientation,
false);

})();
```