Lecture 3: Events, Forms, and Animations

Nick Hynes

January 16, 2014

Topics

1 Events

A GUI without user interaction is just a G! Therefore, the topic of today's class is creating JavaScript applications that respond to input from the user (and look good at the same time)!

How does one do this? With events, of course! Why do I ask so many rhetorical questions? Why is this so self-referential?!? Anyway ... Basically, an event is a notification that is fired when something in the script's environment happens, like mouse clicks or text entry. In JavaScript, an event is represented as an Object that contains properties and methods that allow you to get information about, and interact with, the event.

Given that we have a source, or emitter, of events, all we have to do is add code to "listen" for the appropriate event and call the desired function. The listener doesn't wait around polling for the event, though. Instead, it registers a *callback* function with the browser's event dispatcher that gets called whenever the event is fired. This sort of callback-driven asynchrony is part of what makes JavaScript really unique.

You can also create and dispatch custom events, but, for now, we'll focus on those generated by the browser. Let's start with an example that demonstrates how events work.

Example 1 – Events make me feel bubbly

```
 1  <html>
 2  <!-- omitting head for brevity -->
 3  <body>
 4    <form onclick='console.log("form clicked")'>
 5      <input type='button' onclick='console.log("button clicked")'>
 6    </form>
 7  </body>
 8  </html>
 9
10  When the button is clicked, the console will display:
11  "button clicked"
12  "form clicked"
```

There are a few interesting things happening in this example (none of which are real mouse clicks, however). On lines 4 and 5, we create a form and a button that have their `onclick` attributes set to JavaScript `console.log()` statements; this is how to add event listeners directly to HTML. Additionally, when the button is clicked, both the button and form have their listeners called, in that order. This demonstrates how events *bubble* up the DOM tree.

1.1 Types of events

There are many different kinds of events that the browser emits (see *Additional Resources* for a list (some are really cool!)), but the most important events for the purposes of basic application development (mostly DOM events) fall under the categories of: UIEvent, MouseEvent, KeyboardEvent, InputEvent, FocusEvent, and HashChangeEvent.

UIEvent

Event name	When it's fired
scroll	When the user scrolls the element. The element must have overflow/a scroll bar. The element can be the window or another container element.
resize	When the user resizes the element (usually the window)

The `scroll` event can be attached to the `window`, `document`, or any DOM node and is fired when the user changes the scroll position of the element. If the element does not overflow or have scrollbars, this event will, naturally, never be triggered.
The `resize` event can be attached to the `window` and `document` objects and is fired when either is resized (by changing the browser size or by zooming).

MouseEvent

Event name	When it's fired
click	the user clicks the element. A click is the combination of a mousedown and mouseup.
dblclick	the user double clicks the element
mousedown	the user presses the mouse button on the element
mouseup	the user releases the mouse on an element. Note that the mouse did not have to start on the element.
mouseover	the user hovers the mouse over the element
mouseout	the user stops hovering the mouse over the element
mouseenter	Similar to mouseover except that this event does not bubble
mouseleave	Similar to mouseover except does not bubble

A MouseEvent listener can be attached to any DOM node. The MouseEvent object also has the following properties:

- `screenX`, `screenY` - the mouse's absolute x and y coordinates (relative to screen)
- `clientX`, `clientY` - the mouse's x and y coordinates relative to the top left corner of the page

- `ctrlKey`, `shiftKey`, `altKey`, `metaKey` - booleans that are set if the mouse event occurred while the modifier key was pressed
- `buttons` - The buttons that were pressed during the event. 0 is no button, 1 is the primary button (left), 2 is the secondary button (right), 3 is both 1 and 2, and 4 is the middle button, and so on.

KeyboardEvent

Event name	**When it's fired**
keydown	the user presses a key on the keyboard
keyup	the user releases a key on the keyboard

A KeyboardEvent listener can be attached to the `window` object and any element. If attached to the `window`, the listener will be called if the user either presses or releases a key anywhere in the page. If attached to an element, the KeyboardEvent will only be triggered if the element is focused.

The KeyboardEvent object, like the MouseEvent, also has the following useful properties:

- `key` - a String representing the pressed key (only really useful for control/direction keys)
- `code` - the keycode for the pressed key. This will be the same for all keyboard mappings. The enter key has code 13.
- `ctrlKey, shiftKey, altKey, metaKey` - booleans that are set if a particular modifier key was pressed during the event

Although this is one method of determining if the user has entered text into a form input, the next event is much better suited for the purpose.

InputEvent

Event name	**When it's fired**
input	the user makes a change to an HTML input field or any element that has user-editable content

The InputEvent is generated when the user makes a change to an element that has editable content (e.g. a text box). Thus, the listeners for InputEvents should be attached to editable elements.

FocusEvent

Event name	When it's fired
focus	the user focuses (tabs over to, makes a cursor appear in) an element. This event is dispatched after focus has shifted and does not bubble.
blur	the user removes focus from an element (by focusing another element). This event is dispatched after focus has shifted and does not bubble.
focusin	similar to focus except is dispatched before focus has shifted and does bubble
focusout	similar to blur except is dispatched before focus has shifted and does bubble

Very roughly, an element is focused when the user clicks on it or uses the tab button to select it. Though is usually most useful to attach listeners for FocusEvents to forms and their input elements, any element can be focused and have its FocusEvent listeners triggered.

HashChangeEvent

Event name	When it's fired
hashchange	when `window.location.hash` changes (the part of the URL including and after the #)

Keep in mind that using JS to change `window.location.hash` will result in this event being fired.

1.2 (De-)Registering and triggering event listeners

Registering event listeners

You've already seen one way to register events in the first example. This is done by setting the `onevent` attribute of the element to some JavaScript code.

```
<tagname onevent="JavaScript code">
<!-- where event is the event for which to listen -->
```

Although this is a perfectly valid way of adding event listeners, when you dynamically create elements, it's much better to use one of the following methods since the previous method only allows one listener to be set per event (and also does not support all events).

The preferred way to add an event listener using the DOM API is to call the `addEventListener` function of the selected element. Using jQuery, this most directly maps to the `on` function of the jQuery object called with two arguments.

```
/* DOM API */
element.addEventListener("event", handlerFunction);

/* jQuery */
$jQueryObject.on("event", handlerFunction);

/* where event is the event type, like click, or mousemove,
and handlerFunction is a function that accepts one argument,
the eventObject (can be anonymous) */
//this in the handlerFunction is set to the listening DOM node
```

One thing to keep in mind, or rather, to not keep in mind, is that the jQuery event Object is almost entirely, but not completely like the Event object. More precisely, the jQuery object contains a superset of the properties of the Event object, so they can be used in the same way.

jQuery also has a number of convenience methods that can either trigger or attach a new listener to an element. For example, you have:

- `$jQueryObject.click([handler])` - when called with no arguments, "clicks" on the element or adds `handler` as a listener if present

- `$jQueryObject.focus([handler])` - focuses the element or adds the function `handler` as a focus listener

- see *Additional resources* for a link to more just like these

Example 2 – This example is not uneventful

```
//Creates a button that, when clicked, displays a popup
var aButton = document.createElement("input");
aButton.type = "button";
aButton.value = "Click me!";
aButton.addEventListener("click", function() {
  alert("Thank you for making a simple button very happy.");
});
document.body.appendChild(aButton);

var $aButton = $("<input type='button' value='Click me!'>");
$aButton.click(function () {
  alert("Thank you for making a simple button very happy.");
});
$aButton.appendTo(document.body);
```

De-registering event listeners

De-registering an event listener happens in much the same way as registration (well, as much as doing the exact opposition action can be).

If the listener was attached using the `onevent` property of the element, then simply set the same property to `undefined`. If the listener was attached using the DOM API or jQuery, then use one of the following functions—preferably of the same variety as the one used to register the listener:

```
/* DOM API */
element.removeEventListener("event", handlerFunction);

/* jQuery */
$jQueryObject.off("event", handlerFunction);

/* where event is the event type, like click, or mousemove,
and handlerFunction is a function that accepts one argument,
the eventObject (can be anonymous) */
```

See? They *are* basically the same. One important thing to note is that you can't de-register an anonymous function since the de-registration functions look for, and remove, the Function that is equal to `handlerFunction`. To get around this, either declare a function using the standard `function functionName() {}` syntax or set the function equal to a variable, a la `var functionName = function() {}`.

Triggering event listeners

Before we go on, I want to mention again that `this` in the event handler function is set to the DOM node for which the listener is registered. Okay, now that that's over with, let's continue.

Sometimes you want to call the listeners attached to an element programmatically. There is a way to do this using the DOM API, but it's needlessly complicated and doesn't always work how you'd expect. Fortunately, jQuery allows you to do this in a transparent manner by using the convenience methods listed above, `trigger`, and `triggerHandler`. `trigger` can be a bit more involved so we'll get to that later. `triggerHandler`, on the other hand, is very friendly and easy to use: simply call it on a jQuery object with the event type as a String and it'll call all of the listeners bound to the element for that event type! This doesn't actually dispatch the event, though. Now, as promised, I present to you: `trigger`, a function that creates a real, live event and dispatches it to an element. This can either be called in the same way as `triggerHandler` with the event String or by creating and then calling it with a jQuery event object (the more involved part). Using the jQuery event object is only necessary when dispatching custom events.

```
/* DOM method */
//Left as an exercise to the reader

/* jQuery method */
//To call an element's event listeners
$jQueryObject.triggerHandler("event");

//To dispatch an event to the element
$jQueryObject.trigger("event");
//or
var $jQueryEvent = jQuery.Event("eventname", [properties]);
$jQueryObject.trigger($jQueryEvent);
```

1.3 Things you can do with Event objects

Get the Event's info

The most immediately useful properties of an Event object are `target` and `timestamp`. The former is the DOM node on which the event started (before the event started bubbling) and the latter is the time in milliseconds since the Unix epoch (1/1/1970. On 32-bit systems [using a signed integer], this will occur again on 19/1/2038).

Stop bubbling

When working with an event that bubbles (most of them), it is sometimes desirable to stop the event from bubbling up the DOM tree. For instance, imagine a button that contains a drop-down arrow button; the click handler of the arrow button would need to stop the event from bubbling to prevent the containing button from being clicked.

```
//To stop event bubbling
eventObject.stopPropagation();

//To stop event bubbling and prevent other listeners on the
//same element from being triggered
eventObject.stopImmediatePropagation();
```

Cancel the event

Certain events that have default actions are *cancelable*, meaning that you can prevent the event from doing what it would normally do. This is done by calling the Event's `preventDefault` function.

```
//To cancel an event/prevent its default action
eventObject.preventDefault();
```

Here's a list of cancelable events and what happens when they're canceled:

- keyup, keydown - prevents the key input from being sent to the element or the browser
- mousedown - makes the element unleftclickable (this is totally a word)

It's a bit challenging to script a piece of paper/PDF document to make an example, so please try to use your imagination.

Example 3 – How to use JavaScript to troll your friends

```
//This is unlikely to work in a page that has its own listeners
//You'd have to remove those first
window.addEventListener("mousedown", function(event) {
  event.preventDefault();
  event.stopImmediatePropagation();
  //results in nothing on the page being left-clickable
});
```

The `event.preventDefault()` method also works in jQuery. There are also better ways to cause trouble using JS (like putting an invisible, 100%, fixed div in front of everything, or creating a browser extension that does this). Don't do that though...

2 Forms

An HTML form, created using the `<form>` tag, is a container for user input elements, such as text inputs, checkboxes, buttons, and file uploads. In general, these are created using the void/self-closing `<input>` tag. The type of the input is specified by the element's `type` attribute. The `name` attribute of the item is the name of the data generated by the input (useful for HTTP submitted forms and checkboxes/radio buttons). In the spirit of the rest of the lists and tables in this lecture, here's a table of the more useful input types and a description of each:

Input type	**Description**
text	A text box. Not very exciting, I know, but its metaphorical bread and butter of HTML forms.
password	Like a text box except the entered characters are hidden. For example, in your browser they are likely displayed as dots.
checkbox	A checkbox
radio	A radio (multiple choice, single answer) button. Radio buttons with the same name are considered to be a group in which only one is selectable at a time.
submit	A button that has the text "Submit" and the default action of submitting its parent form.
reset	A button that has the text "Reset" and the default action of resetting its parent form's fields.
button	A button. Does not contain text and has no default action. These can be styled and given click handlers, though.
hidden	A hidden form field. You can use this as a way to store and non-user-editable data to a webpage on form submit.
file	A file select control. Allows the user to select files from the system for upload.
email	A text box that is automatically validated as an email address (not quite as good as making your own regex, though).
url	A text box for URLs that is validated against the regex specified in the `pattern` attribute.
text	A text box. Not very exciting, I know, but its metaphorical bread and butter of HTML forms.

There are also two more input types that are non-void. These are `<textarea>`, which creates a multi-like text input, and `<button>`, which allows you to add content like images and text to the button. The inner HTML of the `<textarea>` is the default value. The `<button>` tag is particularly useful for making styled CSS buttons (an alternative is to style `<a>` elements).

To get/set the contents of text fields/areas, file, and hidden inputs, and set the title of button `input`s and value of checkboxes/radio buttons, just modify the `value` property of the element or use the `val` function of its jQuery object.

```
//To get/set the value/user input of an input-type element
/* DOM API */
var value = element.value;         //getter
element.value = "someValue";       //setter

/* jQuery */
var value = $jQueryObject.val();  //getter
$jQueryObject.val("someValue");   //setter
```

To determine whether a checkbox or radio button is selected, simply query its `checked` property which is a Boolean that indicates its checked status.

```
//To get the value of the checked element (radio or checkbox)
/* DOM API */
var checkables = document.getElementsByName("inputName");
for(var i = 0; i < checkables.length; i++) {
  if(checkables.item(i).checked) {
    console.log(checkables.item(i).value);
  }
}

/* jQuery */
var $checked = $("selector").filter(':checked');
var response = $checked.val();
```

Input elements also have a variety of attributes that allow you to do things like disabling them, making them required, or read-only using the `disabled`, `required`, and `read-only` Boolean attributes. Another new feature that was accomplished using events and whatnot back in the day is the placeholder: now, all you have to do is set the `placeholder` attribute and the browser takes care of the rest. Additionally, there is a `pattern` attribute that validates the form contents against a regular expression.

Example 4 – A formidable form

```
<html>
<!-- head and jQuery omitted for brevity -->
<body>
  <form onsubmit='submitForm()' action='javascript:void(0)'>
    <input type='text' placeholder='Username' id='username'>
    <br>
    <input type='password' placeholder='Password'
    id='password'>
    <br>
    <input type='checkbox' checked='true' id='keepLogged'>
    Keep me logged in.
    <br>
    <input type='submit' value='Submit'>
  </form>
  <script type='application/javascript'>
    function submitForm() {
      var username = $("#username").val();
      var password = $("#password").val();
      var stayLoggedIn = $('#keepLogged').attr("checked");
      doLogin(username, password, stayLoggedIn);

      alert("None shall pass.");
    }
  </script>
</body>
</html>
```

In this example, we create a simple HTML form that contains three input fields; two text boxes for the user's username and password, and a checkbox to give the user the option to stay logged in that is selected by default. There is also a submit button that submits the form when clicked.

The form, itself, has a listener attached to its `submit` event (an event that is unique to forms) that calls the `submitForm` function when the form is (you'll never guess this) submitted. The default action of a form submit is to direct the browser to the URL specified by the `action` property. However, since this is Introduction to JavaScript and not Introduction to PHP, Ruby, Django, or otherwise, we're going to keep the action (no pun intended) on the client side and use a technique known as AJAX to communicate with the server, which we'll cover in the next class. To this end, the `action` attribute is set to the special value `javascript:void(0)`, which is a sort of JavaScript "URL" that does nothing and, thus, prevents the page from reloading.

3 Animations

With the introduction of CSS3, animations became infinitely easier (not to mention better looking!). For historical purposes, I'll provide a quick description of how it was done in ye olde days (circa 2008, but realistically 2011 after it gained more browser support). Essentially, a JavaScript timer was set that incrementally moved a CSS property from one value to another. As you may imagine this was tedious to write and not very efficient, but even then, there were jQuery functions to do just this. In today's shiny, modern browsers, smooth transitions are created simply by setting the CSS `transition` property, changing the desired style, and then watching the browser's native implementation do the rest of the work. This is truly a good world.

3.1 Animation using transitions

To enable transitions on an element, as stated before, simply set its `transition` CSS property. The `transition` property has the following syntax:

```
transition: css-prop duration [timing-function] [delay], ...;
//timing-function and delay are optional parameters

//or use the shorthand notation to animate all CSS properties
transition: all duration [timing-funciton] [delay];
```

Multiple transitioned properties can be achieved by using the `all` keyword or adding more transitions separated by commas.

That's about all there is to CSS animations! Now you're able to create fancy, functional web application GUIs!

3.2 Timing

In JavaScript, there are two main ways to execute code after a certain amount of time, `setTimeout` and `setInterval`. `setTimeout` simply calls a function after the specified time while `setInterval` calls the function repeatedly for every unit of the specified time that passes. While the timing functions are useful for much more than animations, they're included here for their value in creating multi-step animations using transitions!

There is an event for the end of a transition, but it's not fully supported yet. If you'd like to use it, however, the event name is `transitioned` and `webkitTransitionEnd` (which is what I meant by not fully supported).

```
//To call a function after a certain amount of time
var ref = setTimeout(callback, timeInMilliseconds);
clearTimeout(ref); //stops the timeout

//To call a function periodically
var ref = setInterval(callback, timeInMilliseconds);
clearInterval(ref); //stops the interval
```

In the following example you can see how transitions can be combined with timing functions (specifically a recursive `setTimeout`) to create an animation that vaguely resembles a bouncing ball.

3.3 jQuery UI

The jQuery UI extension/library has functions that allow you to trigger simple effects as well as create a variety of widgets (many of which already exist natively in HTML5 (which you should use instead)). You can find a link to documentation on jQuery UI in the *Additional resources* section. However, to pique your interest, I shall provide as an example a snippet that shakes an element up and down.

```
//With jQuery UI imported,
$jQueryObject.effect("bounce"); //Wow, that was easy
```

Example 5 – Not an accurate physical representation

```
<html>
<head>
  <style type='text/css'>
    .bouncingBall {
      transition: top 250ms ease-in;
      width: 100px; height: 100px;
      border-radius: 100px; /* makes a circle */
      position: absolute;
      top: 70%;
      background-color: red;
    }
  </style>
</head>
<body>
  <div class="bouncingBall"></div>
  <!-- jQuery import omitted for brevity -->
  <script type='application/javascript'>
    function bouncyAnimation(maxHeight, goingUp) {
      var $bouncy = $(".bouncingBall");
      maxHeight = maxHeight || $bouncy.css("top");
      if(goingUp) {
        $bouncy.css("top", maxHeight);
        setTimeout(function() {
          bouncyAnimation(maxHeight, false); }, 250);
      } else {
        var screenBottom = window.innerHeight;
        var elemHeight = $bouncy.height();
        $bouncy.css("top", screenBottom - elemHeight);
        setTimeout(function() {
          bouncyAnimation(maxHeight, true); }, 250);
      }
    }
    bouncyAnimation(); //make the magic happen
  </script>
</body>
</html>
```