Events

Mendel Rosenblum

DOM communicates to JavaScript with Events

Event types:

- Mouse-related: mouse movement, button click, enter/leave element
- Keyboard-related: down, up, press
- Focus-related: focus in, focus out (blur)
- Input field changed, Form submitted
- Timer events
- Miscellaneous:
  - Content of an element has changed
  - Page loaded/unloaded
  - Image loaded
  - Uncaught exception

Event handling

Creating an event handler: must specify 3 things:

- What happened: the event of interest.
- Where it happened: an element of interest.
- What to do: JavaScript to invoke when the event occurs on the element.

Specifying the JavaScript of an Event

- Option #1: in the HTML:

```
<div onclick="gotMouseClick('id42'); gotMouse=true;">...</div>
```

- Option #2: from Javascript using the DOM:

```
element.onclick = mouseClick;
    or
element.addEventListener("click", mouseClick);
```

Event object

- Event listener functions passed an `Event` object

  Typically sub-classed `MouseEvent, KeyboardEvent`, etc.

- Some Event properties:

  `type` - The name of the event (`'click', 'mouseDown', 'keyUp'`, ...)

  `timeStamp` - The time that the event was created

  `currentTarget` - Element that listener was registered on

  `target` - Element that dispatched the event

`MouseEvent` and `KeyboardEvent`

- Some `MouseEvent` properties (prototype inherits from `Event`)

  `button` - mouse button that was pressed

  `pageX, pageY`: mouse position relative to the top-left corner of document

  `screenX, screenY`: mouse position in screen coordinates

- Some `KeyboardEvent` properties (prototype inherits from `Event`)

  `keyCode`: identifier for the keyboard key that was pressed
  Not necessarily an ASCII character!

  `charCode`: integer Unicode value corresponding to keypress, if there is one.

Draggable Rectangle - HTML/CSS

```
<style type="text/css">
  #div1 {
    position: absolute;
  }
</style>
...
<div id="div1" onmousedown="mouseDown(event);"
    onmousemove="mouseMove(event);"
    onmouseup="mouseUp(event);">Drag Me!</div>
```

Draggable Rectangle - JavaScript

```
var isMouseDown = false;  // Dragging?
var prevX, prevY;

function mouseDown(event) {
    prevX = event.pageX;
    prevY = event.pageY;
    isMouseDown = true;
}

function mouseUp(event) {
   isMouseDown = false;
}
```

```
function mouseMove(event) {
    if (!isMouseDown) {
      return;
    }
    var elem = document.getElementById("div1");
    elem.style.left = (elem.offsetLeft +
            (event.pageX - prevX)) + "px";
    elem.style.top = (elem.offsetTop +
            (event.pageY - prevY)) + "px";
    prevX = event.pageX;
    prevY = event.pageY;
}
```

Deciding which handler(s) are invoked for an event?

- Complicating factor: elements can contain or overlap other elements

  Suppose user clicks with the mouse on "xyz" in:

```
<body>
  <table>
    <tr>
      <td>xyz</td>
    </tr>
  </table>
</body>
```

  If I have handlers on the `td`, `tr`, `table`, and `body` elements which get called?

  - Sometimes only the innermost element should handle the event
  - Sometimes it's more convenient for an outer element to handle the event

Capturing and Bubbling Events

- Capture phase (or "trickle-down"):
  - Start at the outermost element and work down to the innermost nested element.
  - Each element can stop the capture, so that its children never see the event `event.stopPropagation()`

```
element.addEventListener(eventType, handler, true);
```

- Bubble phase - Most on handlers (e.g. `onclick`) use bubble, not `onfocus/blur`
  - Invoke handlers on the innermost nested element that dispatches the event (mostly right thing)
  - Then repeat on its parent, grandparent, etc. Any given element can stop the bubbling: `event.stopPropagation()`

```
element.addEventListener(eventType, handler, false);
```

- Handlers in the bubble phase more common than capture phase

Example: Timer Events

- Run `myfunc` once, 5 seconds from now:

```
token = setTimeout(myFunc, 5*1000);
```

  Function is called in specified number of milliseconds

- Run `myfunc` every 50 milliseconds:

```
token = setInterval(myfunc, 50);
```

- Cancel a timer:

```
clearInterval(token);
```

- Used for animations, automatic page refreshes, etc.

Event Concurrency

- Events are serialized and processed one-at-a-time
- Event handling does not interleave with other Javascript execution.
  - Handlers run to completion
  - No multi-threading.
- Make reasoning about concurrency easier
  - Rarely need locks.
- Background processing is harder than with threads

Event-based programming is different

- Must wait for someone to invoke your code.
- Must return quickly from the handler (otherwise the application will lock up).
- Key is to maintain control through events: make sure you have declared enough handlers; last resort is a timer.
- Node.js uses event dispatching engine in JavaScript for server programming