More Event Handling

An Aside: Making Users Happy

It's easy to use JavaScript to do annoying things like open popups and alerts, play sounds, prevent users from right-clicking to reach menus, prevent them from scrolling, etc.

As web users, we share common expectations of how websites and browsers should behave and your apps should always aim to fulfill those expectations. The more intuitive your app is, the better.

If a user gets annoyed, they'll just leave your web app or find a way around its limitations. JavaScript is easy to disable through browser settings, so you can't really count on js to force your users to do anything.

At the same time, lots of people are still using old devices and old browsers for various reasons. Don't break the web for them with your fancy or heavy JavaScript. As much as possible, design your apps so that they are usable *to some degree* without JS, and then with older JS features as well. If a certain feature is *absolutely necessary*, don't let your app fail silently: make it clear when JavaScript is required, or when a certain browser version is required, and so on.

```
<noscript>Your browser does not support JavaScript!</noscript>
```

Similarly, make your apps as lightweight and efficient as possible: the less data and resources an app uses, the more devices that can run it. For example, if you listen for the `mousemove` event on the whole window, that will quickly make a browser unresponsive or slow because `mousemove` happens so frequently.

For all the above reasons, we're going to start emphasizing techniques for **cross-browser compatibility** and **graceful degradation**. We've already looked at an example of attaching event listeners in a way that works with both modern browsers and older Internet Explorer (addEventListener versus attachEvent).

The Event Object

Every event triggered in the browser is represented by an object that inherits from the `Event` prototype, such as MouseEvent, KeyboardEvent, etc. How do we get access these event objects?

So far the event handlers we've written are always functions that accept no parameters, like `updateImage` below:

```
function updateImage() {
  var image = document.getElementById("panda");
  image.src = "blah";
}
// attached to DOMContentLoaded
function init() {
  prevButton.addEventListener("click", updateImage);
}
```

When you register an event handler with `addEventListener`, the function you pass in is actually expected to receive the event that has fired as a parameter, but we've just been ignoring that parameter so far. To use the event parameter, we could rewrite `updateImage` as follows:

```
function updateImage(e) {
  // e represents the event that caused this function to be called
  var image = document.getElementById("panda");
  image.src = "blah";
  // example of accessing a property from the Event object
  console.log("Triggered by clicking the following element: " + e.target);
}
// attached to DOMContentLoaded
function init() {
  prevButton.addEventListener("click", updateImage);
}
```

In this example, every time a click event fires, `updateImage` is called with a different event object that represents the click event that just happened. You can access many useful bits of information about the current event from the Event object such as mouse coordinates, the element that was the target of the click, which buttons are pressed and much more.

In the example above, we're using the `target` property, which refers to the HTMLElement that triggered the event, which in this case will be the element referenced by `prevButton`.

This allows you to write event handlers that are much more general: they don't have to be specific to just one kind of event or one target element because you can figure out the context from the event object that your event handler receives as an argument. I can register `updateImage` with many different elements, not just `prevButton`, and then I can use `target` to figure out which of the elements is responsible for the event. Similarly, I could use the `type` property to check whether the event is a click or a kepypress or whatever.

On modern browsers, the event object is passed in as a parameter even if you assign your event handler to the event-handler property, instead of using addEventListener:

```
prevButton.onclick = function (e) {
  // e represents the event that caused this function to be called
  var image = document.getElementById("panda");
  image.src = "blah";
  // example of accessing a property from the Event object
  console.log("Triggered by clicking the following element: " + e.target);
};
```

Cross-browser access to Event object

In the previous section, we only discussed how things work on modern browsers. Let's look again at `attachEvent`, which is what Internet Explorer 8 supports.

`attachEvent` doesn't accept an event object as a parameter; instead, you have to access the most recent event via `window.event`.

```
function updateImage(e) {
  // e is undefined because this handler was registered using attachEvent in IE8
  // so using e.target will raise an error
  console.log("Triggered by clicking the following element: " + e.target);
}

function init() {
  // the Internet Explorer 8 way to do things
  prevButton.attachEvent("onclick", updateImage);
}
```

So, to write an event handler that deals with both possibilities (`attachEvent` versus `addEventListener`), we need to check whether the event parameter is defined. As you see below, we also have to handle an alternative case for the `target` property.

```
function crossBrowserUpdateImage(e) {
  e = e || e.window;
  /* The line above is the same as the following:
    if(typeof e == "undefined") {
      // we're in IE8 so use window.event instead
      e = window.event;
    }
  */
  // Oops, one more discrepancy: in IE 8, e.target is undefined, instead
  // we have e.srcElement
  var target = e.target || e.srcElement;
  console.log("Triggered by clicking the following element: " + target);
}
```

Delaying and Repeating Function Calls with Timers

Roughly speaking, a timer works by queuing an event for the browser to process at a later time. You can use timers to automatically update content in a webpage, like periodically fetching the weather, or hiding a notification after a few seconds.

JavaScript is **single-threaded** meaning that the browser only runs one instruction at a time, so if the browser is busy processing lots of other computationally intensive events, the timer might not finish on time. That is, if you ask for a 2-second timer, *at least* 2 seconds will elapse before the timer completes, but it may also take longer.

You can use `setTimeout` to tell the browser to call a function after some time has ellapsed. This is a one-time timer.

```
// displays an alert after approximately 2 seconds
setTimeout(function() {
  alert('It has been 2000 milliseconds!');
}, 2000);
```

You can use `setInterval` to tell the browser to call a function over and over at regular intervals. This is a recurring timer and it won't stop repeating until you close the web page.

With setInterval, there is no guarantee that your next function call is scheduled to start after the previous one is done. setInterval keeps queueing up more events even if the previously scheduled calls are still in progress. This means you can end up with a big collection of function calls taking turns all at the same time.

The only way to stop `setInterval` is to call `clearInterval` in the code triggered by setInterval.

```
// prints "hello" every second until counter reaches 10
var counter = 0;
var myTimer = setInterval(repeat, 1000);
function repeat() {
  counter++;
  if (counter > 10) {
    clearInterval(myTimer);
  }
  console.log("Hello");
}
```

Similarly, you can call `clearTimeout` to cancel a timer that you started previously with `setTimeout`.

Summary of API used in class

When you manipulate the DOM by retrieving elements with methods like getElementsByTagName, you interact with a hierarchy of objects that include:

- Event
  - target (or srcElement on IE 8)
  - type
- MouseEvent setInterval and clearInterval
- setTimeout and clearTimeout

Last update: 2017-02-13 18:45:02

Source: _javascript_423/lectures/05_more_events.md