# Responding to Events

There are events associated with HTML elements: click, dblclick, mouseover, mouseout, mouseup, mousedown, mousemove, and more. We respond to events by associating functions to them: when an event happens a particular function is called.

## More About Functions

### Defining, Calling, Using Functions

We've already seen that functions can be defined with a *function declaration*:

```
function name() {
  ...
}
name();
```

Function declarations like the above are **hoisted** like `var` statements, so if you declare a function at the bottom of a scope, it's available everywhere. However, it's clearer and safer to declare a function before it is used.

We've also seen the use of *function expressions*, which can be *anonymous*, and that functions can be treated like any other value or object. Here's an anonymous function expression in an assignment statement:

```
// assign a function expression to doSomething
var doSomething = function () {
  ...
}
// call doSomething
doSomething();

// another example: assign the doSomething to the window.onload property
window.onload = doSomething;

// note the difference: this assigns the return value
// of doSomething to window.onload
window.onload = doSomething();
```

Anonymous functions being passed to other functions:

```
document.addEventListener("someevent", function () {
  // function body
});

someFunction = function(item) {
  // function body
}
anArray.map(someFunction)
```

When parenthesis follow a function identifier like `foo()`, that causes the function to be called right away and you end up with its return value. If you want to use the function itself as a value, don't include the parentheses, as in `foo`.

In js, functions are ultimately objects just as numbers, string and arrays are. Therefore functions have properties and function can be treated as values is expressions.

```
// prints true because `now` is indeed a function on the Date object
console.log(typeof Date.now == "function");
// prints false because calling `now` returns a number
console.log(typeof Date.now() == "function");
```

## Function parameters and arguments

When you provide a list of parameters in a function definition, those parameters are always optional.

```
function whatever(a, b, c) {
  //...
}

// All these calls "work"
// a, b, c are all undefined
whatever();
// a, b, c are 1, 2, 3
whatever(1,2,3,4,5,6);
// a is "foo", b is 5, c is undefined
whatever("foo", 5);
```

By default, no error will be thrown if parameter is missing. Similarly, arguments aren't type-checked. So if you want to enforce rules about your function parameters, you have to add error-handling code to check their values.

Note that parameters with a primitive data types (boolean, number, string) are passed *by value*, whereas arrays and other objects are passed by reference.

In the scope of any function, `arguments` is the list of arguments received at run-time. You can access individual arguments with array syntax like `arguments[0]` and use `arguments.length`, but it's important to remember that the `arguments` object is **not an Array**, so other array methods are not available, you can't add elements to it, etc.

```
function whatever(a, b, c) {
  console.log(arguments.length);
;}

// arguments is {}.
// Function prints 0.
whatever();
// arguments is {0:1, 1:2, 2:3, 3:4, 4:5, 5:6}.
// Function prints 6.
whatever(1,2,3,4,5,6);
// arguments is {0:"foo", 1:5}.
// Function prints 2.
whatever("foo", 5);
```

# Event Listeners, Event Handlers

You've seen code like this:

```
function doSomething() {
  //...
}
someElement.addEventListener("click", doSomething);
```

Using formal terms, the above code **registers an event listener** for the `click` event on `someElement`, and it designates the `doSomething` function as the **event handler** for that event. In other words, whenever `someElement` **triggers** a click, `doSomething` will be called.

`addEventListener` is the modern away to register event handlers, and it allows you to register many different event handlers to the same event. However, it's not supported on some old browsers (pre-IE9). We'll look at how to combine `addEventListener` with a fallback method that works with old Internet Explorer.

Note that you can also use `removeEventListener` to deregister an event handler. It's good practice to remove event listeners when they are no longer needed.

## Cross-browser support for addEventListener

Old versions of Internet Explorer don't recognize `addEventListener` and instead use a different function called `attachEvent` that does the same thing. In old IE, you'd so something like `myElement.attachEvent("onclick", doSomething)`.

Here is an example of a function for adding event listeners in a way that works with both new and old browser versions.

```
function addEvent(obj, type, fn) {
  // check whether obj exists and whether obj has addEventListener
  if (obj && obj.addEventListener) {
    // modern browser
    obj.addEventListener(type, fn, false);
  } else if (obj && obj.attachEvent) {
    // Older IE
    obj.attachEvent("on" + type, fn);
  }
}
// Example usage
addEvent(window, "load", doSomething);
```

## Another way to register event listeners: event handler properties

Built-in browser objects like `Window`, `Document` and `HTMLElement` all have properties to designate specific event handlers. These properties have names like `onclick`, `onload`, `onsubmit`, and so on.

```
// Register doSomething as the event handler for the window.load event
window.onload = doSomething;
// You can also remove an event handler
window.onload = null;
```

This technique is simple and clear, and works well across all major browsers because it's been around for a long time. On the other hand it limits you to only one event handler per event.

```
myElement.onclick = doSomething;
// this replaces doSomething with doAnotherThing as the event handler
myElement.onclick = doAnotherThing;
```

In fact, event handler properties can lead to some tricky bugs: it's easy to accidentally overwrite one event handler with another when using event handler properties. One js file you have could set onclick one way, and another js file could overwrite it.

### How *not* to handle events: inline event handlers

The following technique is old and **should no longer be used**. It consists of adding an event-handler attribute like onclick directly to an HTML tag.

```
<a href="somepage.html" onclick="doSomething();">Some Link</a>
```

This has fallen out of favour because it creates a messy mix of HTML and JavaScript and it makes it harder to create web apps that behave well when JavaScript is disable by the user.

## Kinds of Events

This next section highlights some most common events. Note that some of these events are not necessarily supported in the same way (or at all) in different browsers or on different devices. Consider how possible events might be different on a mobile device, or a screen reader.

### Pointer events

You can observe how these events get triggered at this demo of mouse events

- mousedown
- mouseup
- **click**: a combination of mousedown and mouseup on the same elements. On some browsers this is only triggered for the left mouse button.
- drag: mousedown on one element, mouse up on another element
- dblclick
- contextmenu: right-clicking (windows, linux) or ctrl+clicking (mac)
- **mouseover**
- **mouseout**
- mousemove: so many mousemoves happen with even a small motion of the mouse that listening to this event can easily cause performance issues. If you must use this, apply it to a small area of the page and detach the listener as soon as you don't need it.

Reflection: what happens when you attach different event listeners for both the click event and the dblclick event on the same element?

### Keyboard events

You can observe how these events get triggered at this demo of keyboard events

- keydown: pressing a key. When you press and hold a key, multiple keydown events will be emitted.
- keyup: releasing a key.

- keypress: a combination of keydown and keyup (only for character[1] keys on some browsers). See keypress is bananas

These events are usually observed on input elements (forms), but you can also listen for them on a canvas element or an entire document. They are tied to actually pressing on the keyboard; that is, if you use your mouse to paste text into a form, no key events will be triggered.

Key detection can vary across browsers, especially when you get to certain Unicode characters like emoji or mathematical symbols.

## Browser Events

## Form Events

---

1. A character key is a key that produces a symbol. Tab, arrows, ctrl, shift are not considered character keys. Of course, this varies by browser.↩

---

Last update: 2017-02-07 21:53:31

Source: _javascript_423/lectures/04_events.md