Javascript Madness Intro

JavaScript Madness: Keyboard Events

Jan Wolter

Note: I have stopped updating this page. At this point nearly all popular browsers are have achieved a good level of compatibility on most of these features, and their behavior with respect to them just isn't changing much anymore. The only thing web designers really need to still watch out for is IE8, which is lingering due to the fact that it is the best version of IE that works on windows XP.

1. Introduction

This document summarizes the results of some browser tests done while attempting to implement key stroke handling code in JavaScript. It documents inconsistancies in the way different browsers implement keyboard events.

The tests were originally done with the intention of learning just enough to write the code I needed to write. Coverage has expanded considerably since then, but the results here still are not comprehensive or authoritative and do not cover all aspects of keyboard event handling.

This data is based on tests of many, many browsers over many, many years, but is not comprehensive. I update it periodically as new browsers cross my desktop. The browser versions most recently tested are:

	Windows	Macintosh	Linux
Internet Explorer	9.0.8112.16421	5.2	-
Firefox	13.0.1 (Gecko 13.0)	5.0.1 (Gecko 5.0.1)	4.0 (Gecko 2.0)
Safari	5.1.7 (WebKit 534.57.2)	5.0.2 (WebKit 533.18.1)	-
Chrome	20.0.1132.57 (WebKit 536.11)	-	4.0.249.43 Beta (WebKit 532.5)
Opera	11.52	9.10	10.10
Konqueror	-	-	4.3.1

The script used to collect the test results reported here is available at http://unixpapa.com/js/testkey.html. I mostly report only what I can test myself, so this report is necessarily incomplete:

- It primarily focuses on standard US keyboards. There are a huge range of other keyboard layouts in use in the world, which include not only different characters, but standard characters in different places. So, for example, many UK keyboards have a 3 £ key and a # ~ key, neither of which exists on US keyboards. I don't know what keycodes keys like these send.

- It does not cover the behavior of keypad keys on the Macintosh, because none of my Macs have keypads.

This document will usually refer to "Gecko" instead of "Firefox" and to "WebKit" instead of "Safari" or "Chrome". That's because browser behavior usually depends on the rendering engine, and different browsers that use the same rendering engine work the same. See the Layout Engine page for more information, including mappings of layout engine versions to browser versions.

Previous versions of this document included coverage of the iCab 3 browser, but iCab has switched to using the WebKit rendering engine, and so presumably behaves exactly like Safari. Since it is unlikely that many web developers will want to go out of their way to support iCab 3, that material has been removed from this document and archived in a separate report on iCab 3.

2. Event Triggering

In all recent browsers, pressing a key triggers a series of Javascript events which can be captured and handled. These events, however, were not defined by any standard until DOM3 which few browsers have yet implemented.

There is strong agreement across all browsers about which events should be sent and what order they should be sent in when a key is pressed:

Browser	*Events sent when normal key is typed*
All Browsers	**keydown** **keypress** **keyup**

Windows versions of Opera have a bit of buggy behavior: when you type the +, -, *, or / keys on the *keypad*, then two **keypress** events are triggered instead of one. This has been observed on Opera 11 and Opera 8.5. I don't know how long this bug has been around.

The **keydown** event occurs when the key is pressed, followed immediately by the **keypress** event. Then the **keyup** event is generated when the key is released.

To understand the difference between **keydown** and **keypress**, it is useful to distinguish between "characters" and "keys". A "key" is a physical button on the computer's keyboard. A "character" is a symbol typed by pressing a button. On a US keyboard, hitting the 4 key while holding down the Shift key typically produces a "dollar sign" character. This is not necessarily the case on every keyboard in the world. In theory, the **keydown** and **keyup** events represent keys being pressed or released, while the **keypress** event represents a character being typed. In practice, this is not always the way it is implemented.

For a while, some browers fired an additional event, called **textInput**, immediately after **keypress**. Early versions of the DOM 3 standard intended this as a replacement for the **keypress** event, but the whole notion was later revoked. Webkit supported this between versions 525 and 533, and I'm told IE supported it, but I never detected that, possibly because Webkit required it to be called `"textInput"` while IE called it `"textinput"`.

There is also an event called **input**, supported by all browsers, which is fired just after a change is made to to a textarea or input field. Typically **keypress** will fire, then the typed charater will appear in the text area, then **input** will fire. The **input** event doesn't actually give any information about what key was typed - you'd have to inspect the textbox to figure it out what changed - so we don't really consider it a key event and don't really document it here. Though it was originally defined only for textareas and input boxes, I believe there is some movement toward generalizing it to fire on other types of objects as well.

2.1. Events Triggered by Special Keys

In addition to all the normal keys used to input ASCII characters, keyboards typically have many special purpose keys that do other things. These do not necessarily generate the same events as normal keys, and they show less consistency across browsers.

"Modifier keys" are one class of special keys. They include keys like Shift, Control and Alt, that don't send characters, but modify the characters sent by other keys. For nearly all modern browsers, both **keydown** and **keyup** events are triggered by modifier keys, but **keypress** events are not. This is consistant with their being "key" events not "character" events.

However, Konqueror and some older browser versions do have different behaviors:

Browser	*Events sent when modifier keys are typed*
Gecko ≥ 1.7 Internet Explorer WebKit ≥ 525 Opera ≥ 10.10	**keydown** **keyup**
Opera ≤ 9.50 Konqueror	**keydown** **keypress**

	keyup
WebKit < 525 Gecko 1.6	no events sent

Most browsers treat the Caps Lock key the same as any other modifier key, sending **keydown** when it is depressed and **keyup** when it is released, but there are exceptions. Older Gecko browsers generated a **keypress** event for Caps Lock, but that has been fixed. Macintosh versions of Safari 3 get really clever: each time you strike and release the Caps Lock key, only one event is triggered, and it is **keydown** if you turning on caps-lock mode and **keyup** if you are turning it off. Safari does not do this with Num Lock.

There are many other special keys on a typical keyboard that do not normally send characters. These include the four arrow keys, navigation keys like Home and Page Up, special function keys like Insert and Delete, and the function keys F1 through F12. Internet Explorer and WebKit 525 seem to classify all of these with the modifier keys, since they generate no text, so in those browsers there is no **keypress** event for them, only **keyup** and **keydown**. Many other browsers, like Gecko, do generate **keypress** events for these keys, however.

Old versions of WebKit had a bug that caused two identical **keyup** events to be triggered when arrow keys and other special keys were released. I know this existed in WebKit 312 and I know it was fixed in WebKit 525, but I don't know when it was fixed.

Standard Windows keyboards typically have two Start keys and a Menu key, while Apple keyboards have two Apple keys. I'm not going to attempt to describe the behavior of those keys in detail here. They are very inconsistent across browsers, don't exist on all keyboards, and they frequently have default actions that cannot be disabled. As such, Javascript programmers would be well advised to stay away from them.

If NumLock is off, and you hit keypad number key while holding Shift down, then Windows systems trigger some extra events. Windows browsers pretend that the Shift key was released before the key was typed, and then pressed again after it was released, and they trigger **keyup**, **keydown** and (in some browsers) **keypress** events to indicate this. Linux systems don't do this. I don't know if Macintoshes do.

2.2. Events Triggered on Auto-Repeat

If a key is held down long enough it typically auto-repeats, and some additional events will be triggered on each autorepeat. On Macintosh and Linux systems, modifier keys usually don't auto-repeat, but on Windows systems they do (which seems weird to me). In most browsers, an autorepeat is sensibly treated as a character event, but not a key event, so it triggers a **keypress** but not a **keydown** or **keyup**. But, of course, there is some variation:

Browser	*Events triggered on each autoreapeat*	
	normal keys	*special keys*
Internet Explorer (Windows) WebKit ≥ 534	**keydown** **keypress**	**keydown**
525 ≤ WebKit ≤ 533	**keydown** **keypress** **textInput**	**keydown**
Gecko (Windows)	**keydown** **keypress**	
Gecko (Some Linuxs) Gecko (Macintosh) WebKit < 525 Konqueror Opera	**keypress** only	
Gecko (Oher Linuxs)	**keyup** **keydown** **keypress**	
Internet Explorer (Macintosh)	*no events triggered*	

Gecko's behavior seems to be different on different versions of Linux. On some versions of Linux, mostly newer versions, it

generates extra events, in a manner only previously seen on iCab 3. I don't know exactly what makes the difference.

2.3. Suppressing Default Event Handling

If you are installing your own handlers for key events, then sometimes you won't want the browser default action to occur (such as having the character appear in a text entry area). To prevent this, you typically have the event handler return `false`, and maybe call `event.preventDefault()` and `event.stopPropagation()` if they are defined. But on which event handler must you suppress defaults? This, of course, varies from browser to browser.

Browser	*Which event handlers need to suppress defaults to prevent key from appearing in text box*
Internet Explorer Gecko Webkit Opera ≥ 11.52 Konqueror 4.3	either **keydown** or **keypress**
Opera ≤ 10.53 Konqueror 3.5	**keypress**
Konqueror 3.2	**keydown**

Suppressing defaults on the **keydown** event has some odd side effects on some browsers, in that it may prevent some other events from firing. Apparantly, triggering further events is taken to be part of the default action of the **keydown** event in these browsers.

Browser	*Side effect suppressing defaults on **keydown***
Gecko WebKit < 525 Opera	No change
Internet Explorer WebKit ≥ 525	**keypress** event never occurs. **keyup** event works normally.
Konqueror	**keypress** event only occurs on auto repeats. **keyup** event works normally.

In versions of WebKit and IE that supported **textInput**, that event would not fire if **keydown** or **keypress** suppressed defaults. Suppressing defaults on that event would prevent the typed character from appearing in input boxes.

In Konqueror 4.3.1, I noticed a brand new weirdness. If you don't suppress the default action on **keyup** then you get two **keyup** events. I also seemed to sometimes get duplicate **keydown** and **keypress** events if defaults weren't suppressed on either **keydown** or **keypress**.

Most applications will either use only **keypress** or use only **keyup/keydown**, so this all works out pretty well in most browsers. If you are handling **keypress** and want to suppress default handling of the key, return `false` from that handler. If you are handling **keydown/keyup** and want to suppress defaults, install a **keypress** handler that does nothing except return false.

2.4. Event Triggering Summary

To give a clearer side by side comparison, suppose we press the Shift key, then press the A key, holding it down long enough to auto-repeat just once, then release A, and the release Shift. The events we see are shown below for various browsers. Events marked in red do not occur if there is a **keydown** handler that returns `false`.

	Internet Explorer (Windows) WebKit≥534	Gecko (Windows)	Gecko (Linux/Mac) Opera≥10.10	525≤WebKit≤533	WebKit<525	Opera≤9.50	Konqueror	Internet Explorer (Mac)
Shift pressed	**keydown**	**keydown**	**keydown**	**keydown**		**keydown** **keypress**	**keydown** **keypress**	**keydown**
				keydown				

A pressed	**keydown keypress**	**keydown keypress**	**keydown keypress**	**keypress textInput**	**keydown keypress**	**keydown keypress**	**keydown keypress**	**keydown keypress**
A autorepeats	**keydown keypress**	**keydown keypress**	**keypress**	**keydown keypress textInput**	**keypress**	**keypress**	**keypress**	
A released	**keyup**	**keyup**	**keyup**	**keyup**	**keyup**	**keyup**	**keyup**	**keyup**
Shift released	**keyup**	**keyup**	**keyup**	**keyup**		**keyup**	**keyup**	**keyup**

I used to exclaim here about no two browsers being alike here, but progress is being made. The newer versions of WebKit are identical to IE and Opera is identical to Linux/Mac versions of Gecko.

3. Identifying Keys

When you catch a keyboard event, you may wish to know which key was pressed. If so, you may be asking too much. This is a very big mess of browser incompatibilities and bugs.

3.1. Classic Values Returned on Key Events

The **keydown** and **keyup** events should return a code identifying a key, not a code identifying a character. It is not obvious how to do this. ASCII codes don't really suffice, since the same key can generate different characters (if combined with shift or control), and the same character can be generated by different keys (such as the numbers on the keyboard and the numbers on the keypad). Different browsers use different ways of assigning numeric codes to the different keys. We will call these "Mozilla keycodes", "IE keycodes", "Opera keycodes" and "psuedo-ASCII codes" and we'll explain them in more detail below.

Not only do the browsers differ in what values they return, they differ in where they return them. Three different properties of the event object may be used to return them. They are `event.keyCode`, `event.which` and `event.charCode`.

keydown and keyup events			
	`event.keyCode`	`event.which`	`event.charCode`
IE <9.0 (Windows)	IE keycode	undefined	undefined
Internet Explorer (Mac)	IE keycode	undefined	extended ASCII code
IE ≥ 9.0 WebKit ≥ 525	IE keycode	IE keycode	zero
WebKit < 525	IE keycode	IE keycode	ASCII code if ASCII character, zero otherwise
Gecko	Mozilla keycode	Mozilla keycode	zero
Opera ≥ 9.50 (all platforms) Opera 7 (Windows)	Mozilla keycode except keypad and branded keys give Opera keycodes	Mozilla keycode except keypad and branded keys give Opera keycodes	undefined
Opera 8.0 to 9.27 (Windows)	Opera keycode	Opera keycode	undefined
Opera < 9.50 (Linux & Macintosh)	Pseudo-ASCII code	Pseudo-ASCII code	undefined
Konqueror 4.3	Pseudo-ASCII code	Pseudo-ASCII code	zero
Konqueror 3.5	Pseudo-ASCII code	Pseudo-ASCII code if key has an ASCII code, zero otherwise	zero

Konqueror 3.2	Pseudo-ASCII code	Pseudo-ASCII code	undefined

In version 9.50, Opera abandoned Opera keycodes and Pseudo-ASCII keycodes in favor of Mozilla keycodes for most keys (thus reverting to the behavior of Windows Opera 7). WebKit has modified their Konqueror-derived code to use IE keycodes, and I expect Konqueror will follow. Thus there seems to be a convergences on the IE and Mozilla keycodes, which are pretty similar. This is kind of encouraging.

On **keydown** and **keyup**, the event objects also have flags that indicate which modifier keys were being pressed when the key was typed. These are:

```
event.shiftKey
event.ctrlKey
event.altKey
event.metaKey
```

These all have `true` or `false` values. According to the DOM 3 standard, on the Macintosh, the Option key should activate `event.altKey` and the Command key should activate `event.metaKey`. These attributes seem to work correctly on all modern browsers tested, except `event.metaKey` is undefined in all versions IE. There is some freakishness in obsolete browsers that can probably be ignored these days. In Macintosh versions of IE, the Command key sets `event.ctrlKey` and the Control key does nothing. In Netscape 4, none of these attributes existed and the `event.modifiers` attribute needed to be used instead.

One would think that if a key is typed when Caps Lock is on, then `event.shiftKey` would be `true`, but this is not the case in any browser tested. There is also a lot of inconsistency in the values these flags take on the **keydown** and **keyup** events actually associated with pressing and releasing the modifier keys, but I can't imagine anyone would care enough to justify documenting the details.

3.2. Classic Values Returned on Character Events

For **keypress** events, it is pretty clear that the ASCII code of the typed character should be returned, and pretty much all browsers do that.

But what if there is no ASCII code associated with the key? Arrow keys and keys like Page Down and F1 don't have ASCII codes. We call these "special" keys in contrast to the "normal" keys that have ASCII codes. Note that Esc, Backspace, Enter, and Tab are "normal" because they have ASCII codes.

When **keypress** events are generated for special keys, the browser needs to return some non-ASCII value to indicate which key ways pressed. We'll see that various different browsers do this in different ways.

Some browsers avoid this problem by not generating **keypress** events for special keys. A good case can be made that this is the right thing to do, since these keystrokes are arguably not character events. But such arguments are weakened by the arbitrariness of the division between normal and special keys. Why should the keyboard Backspace key have a **keypress** event, but not the keypad Delete key? Is Tab really fundamentally different than right arrow?

keypress events				
		`event.keyCode`	`event.which`	`event.charCode`
IE < 9.0 (Windows)	normal:	ASCII code	undefined	undefined
	special:	*no keypress events for special keys*		
IE (Mac)	normal:	ASCII code	undefined	ASCII code
	special:	*no keypress events for special keys*		
Gecko	normal:	zero	ASCII code	ASCII code
	special:	Mozilla keycode	zero	zero
IE ≥ 9.0 WebKit ≥ 525	normal:	ASCII code	ASCII code	ASCII code
	special:	*no keypress events for special keys*		
WebKit < 525	normal:	ASCII code	ASCII code	ASCII code
	special:	extended ASCII code	extended ASCII code	extended ASCII code

Opera ≥ 10.50 (all platforms)	normal:	ASCII code	ASCII code	undefined
	special:	Mozilla keycode, except keypad and branded keys give Opera keycodes	zero	undefined
Opera ≥ 9.50 (all platforms) Opera 7 (Windows)	normal:	ASCII code	ASCII code	undefined
	special:	Mozilla keycode, except keypad and branded keys give Opera keycodes	zero for arrows, function keys, PageUp, PageDown same as `event.keyCode` otherwise	undefined
Opera 8.0 to 9.27 (Windows)	normal:	ASCII code	ASCII code	undefined
	special:	Opera keycode	zero for arrows, function keys, PageUp and PageDown, same as `event.keyCode` otherwise	undefined
Opera < 9.50 (Linux & Macintosh)	normal:	ASCII code	ASCII code	undefined
	special:	Opera keycode	zero for arrows, function keys, PageUp and PageDown, same as `event.keyCode` otherwise	undefined
Konqueror 4.3	normal:	ASCII code	ASCII code	ASCII code
	special:	Pseudo-ASCII code	Pseudo-ASCII code	zero
Konqueror 3.5	normal:	ASCII code	ASCII code	ASCII code
	special:	Pseudo-ASCII code	zero	zero
Konqueror 3.2	normal:	ASCII code	ASCII code	undefined
	special:	*no keypress events for special keys*		

The traditional method to distinguish special keys from normal keys on **keypress** events is to first check `event.which`. If it is undefined or non-zero, then the event is from a normal key, and the ASCII code for that key is in `event.keyCode`. If it is defined as zero, the the event is from a special key, and the keycode is in `event.keyCode`. This works for almost every browser, but there are two exceptions:

- The newest version of Konqueror that I have tested, version 4.3.1, returns non-zero `event.which` values for all special keys. The only way to distinguish an up arrow from an ampersand is to check `event.charCode`.
- Versions of Opera before 10.50 messes up by returning non-zero `event.which` values for four special keys (Insert, Delete, Home and End).

So, I guess with this new botched version of Konqueror, we have to make our tests more complex. If neither `event.which` nor `event.charCode` is defined as zero, then it is a normal key event.

The DOM 3 standard makes a half-hearted attempt to suggest standards for these "legacy" attributes. Konqueror's annoying non-zero `event.which` values for special keys actually kind of comply with what it suggests. Except by that standard, **keypress** shouldn't be firing at all for special keys.

If you are actually interested in special key events, then probably you should be hooking your code into **keydown** and **keyup**, which work more consistently across browsers. So the main practical importance of this is that **keypress** handlers should not treat `event.keyCode` as an ASCII code if either `event.which` or `event.charCode` is defined as zero.

The flags `event.shiftKey`, `event.ctrlKey`, `event.altKey` and `event.metaKey` are typically defined on **keypress** events, just as they are on **keydown** and **keyup** events. WebKit seems to define `event.keyIdentifier` on **keypress** as well, but I wouldn't count on future browsers doing that.

On **textInput** events, `event.data` contained the text that was input. On key inputs, this was typically a one character string. Since there were no **textInput** events on special keys, we didn't have to worry about such cases.

3.3. Key Code Values

Now for the actual values being returned for different keys. Some people refer to the Mozilla/IE keycodes as "scan codes". Scan codes are returned from the keyboard, and are converted to ASCII by the keyboard drivers. They typically vary with different kinds of keyboards. As far as I can tell, these are NOT scan codes. They vary with the browser type rather more than with keyboard type, and they don't seem to match with any keyboard scan codes that I've seen documented.

The table below lists values for keys commonly found on US keyboards. If a single value is given, then that value is sent whether the Shift key is held down or not. If two values *x*/*y* are given, the the first is sent when the key is unshifted, and the second is sent when the key is shifted. (Ideally there should be slashed values only in the ASCII column of this table, since the other codes are all used only on **keyup** and **keydown** events, which are key events not character events.)

Keys highlighted in green are consistent across all browsers tested. Keys highlighted in yellow are consistent for recent versions of IE, Gecko, WebKit and Opera. Keys highlighted in red aren't.

Key	ASCII	Mozilla keycodes	IE keycodes	Opera keycodes	pseudo ASCII codes	exceptions
Alphabetic keys A to Z	97/65 to 122/90	ASCII code of uppercase version of the letter 65 to 90				
Space	32	32	32	32	32	
Enter	13	13	13	13	13	
Tab	9	9	9	9	9	
Esc	27	27	27	27	27	
Backspace	8	8	8	8	8	
Modifier Keys						
Key	**ASCII**	**Mozilla keycodes**	**IE keycodes**	**Opera keycodes**	**pseudo ASCII codes**	**exceptions**
Shift	-	16	16	16	16	**Linux Opera < 9.0**: 0
Control	-	17	17	17	17	**Linux Opera < 9.0**: 0 **Mac Opera**: 0
Alt	-	18	18	18	18	**Linux Opera < 9.0**: 0
Caps Lock	-	20	20	20	20	**Linux Opera**: 0
Num Lock	-	144	144	144	144	**Linux Opera < 9.50**: 0 **Win Opera < 9.00**: 0
Keyboard Number Keys						
Key	**ASCII**	**Mozilla keycodes**	**IE keycodes**	**Opera keycodes**	**pseudo ASCII codes**	**exceptions**
1 !	49/33	49	49	49	49/33	**Mac Gecko < 1.8**: 49/0
2 @	50/64	50	50	50	50/64	**Mac Gecko < 1.9**: 50/0
3 #	51/35	51	51	51	51/35	**Mac Gecko < 1.9**: 51/0
4 $	52/36	52	52	52	52/36	**Mac Gecko < 1.9**: 52/0
5 %	53/37	53	53	53	53/37	**Mac Gecko < 1.9**: 53/0
6 ^	54/94	54	54	54	54/94	**Mac Gecko < 1.9**: 54/0
						Mac Gecko < 1.9:

7 &	55/38	55	55	55	55/38	55/0
8 *	56/42	56	56	56	56/42	**Mac Gecko < 1.9**: 56/0
9 (	57/40	57	57	57	57/40	**Mac Gecko < 1.9**: 57/0
0)	48/41	48	48	48	48/41	**Mac Gecko < 1.9**: 48/0
Symbol Keys						
Key	**ASCII**	**Mozilla keycodes**	**IE keycodes**	**Opera keycodes**	**pseudo ASCII codes**	**exceptions**
; :	59/58	59	186	59	59/58	**Mac Gecko**: 59/0
= +	61/43	61	187	61	61/43	**Mac Gecko ≥ 1.9**: 61/107 **Mac Gecko < 1.9**: 61/0
, <	44/60	188	188	44	44/60	**Mac Gecko**: 188/0
- _	45/95	109	189	45	45/95	**Mac Gecko ≥ 1.9**: 109/0 **Mac Gecko < 1.9**: 0
. >	46/62	190	190	46	46/62	**Mac Gecko**: 190/0
/ ?	47/63	191	191	47	47/63	**Mac Gecko**: 191/0
` ~	96/126	192	192	96	96/126	**Mac Gecko**: 192/0
[{	91/123	219	219	91	91/123	
\ |	92/124	220	220	92	92/124	**Mac Gecko**: 220/0
] }	93/125	221	221	93	93/125	
' "	39/34	222	222	39	39/34	
Arrow Keys						
Key	**ASCII**	**Mozilla keycodes**	**IE keycodes**	**Opera keycodes**	**pseudo ASCII codes**	**exceptions**
left-arrow	-	37	37	37	37	
up-arrow	-	38	38	38	38	
right-arrow	-	39	39	39	39	
down-arrow	-	40	40	40	40	
Special Keys						
Key	**ASCII**	**Mozilla keycodes**	**IE keycodes**	**Opera keycodes**	**pseudo ASCII codes**	**exceptions**
Insert	-	45	45	45	45	**Konqueror**: 0 **Opera < 9.0**: 0
Delete	-	46	46	46	46	**Konqueror**: 127 **Opera < 9.0**: 0
Home	-	36	36	36	36	**Opera < 9.0**: 0
End	-	35	35	35	35	**Opera < 9.0**: 0
Page Up	-	33	33	33	33	
Page Down	-	34	34	34	34	
Function Keys F1 to F12	-	112 to 123	112 to 123	112 to 123	112 to 123	
Keypad Keys If Num Lock is on, unshifted/shifted values are returned as shown below. If Num Lock is off, Linux browsers						

reverse the shifted/unshifted values, while Windows browsers always return the shifted value. None of my Macintoshs have a keypad, so I don't know what they do.

Key	ASCII	Mozilla keycodes	IE keycodes	Opera keycodes	pseudo ASCII codes	exceptions
. Del	46/-	110/46	110/46	78/46	78/46	**Opera < 9.0**: 78/0 **Linux Opera 11.5**: 190/46
0 Ins	48/-	96/45	96/45	48/45	48/45	**Opera < 9.0**: 48/0
1 End	49/-	97/35	97/35	49/35	49/35	**Opera < 9.0**: 49/0
2 down-arrow	50/-	98/40	98/40	50/40	50/40	
3 Pg Dn	51/-	99/34	99/34	51/34	51/34	
4 left-arrow	52/-	100/37	100/37	52/37	52/37	
5	53/-	101/12	101/12	53/12	53/12	**Linux Opera**: 53/0
6 right-arrow	54/-	102/39	102/39	54/39	54/39	
7 Home	55/-	103/36	103/36	55/36	55/36	**Opera < 9.0**: 55/0
8 up-arrow	56/-	104/38	104/38	56/38	56/38	
9 Pg Up	57/-	105/33	105/33	57/33	57/33	
+	43	107	107	43	43	**Linux Opera 11.5**: 61
-	45	109	109	45	45	**Linux Opera 11.5**: 109
*	42	106	106	42	42	**Linux Opera 11.5**: 56
/	47	111	111	47	47	**Linux Opera 11.5**: 191
Keypad Enter	13	13	13	13	13	

Branded Keys

Key	ASCII	Mozilla keycodes	IE keycodes	Opera keycodes	pseudo ASCII codes	exceptions
Left Apple Command	-	224	?	17	?	**WebKit ≥ 525**: 91
Right Apple Command	-	224	?	17	?	**WebKit ≥ 525**: 93
Left Windows Start	-	91	91	219	0	**Linux Gecko**: 0
Right Windows Start	-	92	92	220	0	**Linux Gecko**: 0
Windows Menu	-	93	93	0	0	

Note that all four encodings agree on most of the common keys, the ones highlighted in green in this table. For the letters and numbers and for spaces, tabs, enters, and arrows the codes are all the same. In fact, they are all standard ASCII values (except for the arrows).

For symbols, things are a fair mess. IE and Mozilla don't entirely agree on what the codes should be. Three keys, ; :, = + and - _, have different values in IE and Mozilla keycodes. Furthermore, there are long standing bugs in Macintosh versions of Gecko that have caused zero keyCodes to be returned for many symbols.

The Opera keycodes have been abandoned by Opera, but they had a certain simple charm. They were always the ASCII code of the character that the key sends when it is not modified by shift or control. They don't allow you to distinguish numbers typed on the keypad from numbers typed on the keyboard, and such like things, but they are, at least, fairly intuitive.

The pseudo ASCII codes weren't really keycodes at all. They were just the ASCII code for the character *except* that for lower

case letters the upper case ASCII code is sent. So those browsers really entirely abandoned the idea of keycodes, instead returning character codes slightly modified for partial IE compatibility. There is much to be said for abandoning keycodes, since the concept really gets you in trouble as you try to handle international keyboards, but something is lost when you do that. You can't, for example, tell if a number was typed on the main keyboard or the keypad. I prefer WebKit's approach, where they keep the keycodes (making them entirely compatible with IE keycodes) but *also* return the character code on all key events.

Using pseudo-ASCII codes causes another problem: you can't always recognize the arrow keys on **keydown** and **keyup** events. These browsers send the same codes as IE does for arrow keys: the values 37, 38, 39, and 40. These happen to be the ASCII codes for "`%`", "`&`", "`'`" and "`(`". On U.S. keyboards all those five characters are sent by shifted keys, so you'll never see them as keycodes under any of the three keycode schemes. (Some foreign keyboards do create these characters from unshifted keys, but I don't know what keycodes are sent by those keys.) But when pseudo-ASCII keycodes are used these same values are also sent when you type those keys, so you can't tell those symbols from arrow keys. Similar problems occur with some of the other special keys like Home which sends the same values as "`$`".

For browsers that generate **keypress** events for special keys, it is also generally true that `event.keyCode` will have the same value for "left-arrow" and "%", however we can usually tell which it is because `event.which` is zero for special keys (there are problems with this in Opera and Konqueror, see above). Versions of WebKit before 525 took a different approach. They invented unique values to return instead of ASCII codes for special keys, and returned the same value in `event.keyCode`, `event.which`, and `event.charCode`. The table below gives the extended ASCII codes returned by old WebKit versions, and also the ones returned in `event.charCode` on **keydown** and **keyup** events in Macintosh versions of IE.

key	*Extended ASCII codes for Special Keys*								
	up arrow	down arrow	left arrow	right arrow	function keys F1 to F12	Home	End	Page Up	Page Down
WebKit < 525	63232	63233	63234	63235	63236 to 63247	63273	63275	63276	63277
Macintosh IE	30	31	28	29	16 for all keys	*no events triggered*			

To complete the thoroughness of the mess, keycode generation in current Macintosh versions of Gecko remains buggy. For many keys, no keycodes are returned on **keydown** and **keyup** events. Instead the `keyCode` value is just zero. Some of these problems were fixed in Gecko 1.9, but not all, and the keyboard plus key started returning the value that is supposed to be returned by the number pad plus key.

characters	*Keycodes on Gecko **keyup** and **keydown** events*		
	Linux and Windows Gecko (correct)	Macintosh Gecko 1.8 and older (buggy)	Macintosh Gecko 1.9 and later (buggy)
! @ # $ % ^ & * ( )	Same as number keys these symbols appear on	zero	Same as number keys these symbols appear on
-	109	zero	109
_ ~ | < > ? :	Same as unshifted symbol keys these symbols appear on	zero	zero
+	61	zero	107
Any key typed with ALT key held down	Same code as without ALT key	zero	Same code as without ALT key

Macintosh Gecko does give correct charCode values on **keypress** events, but to a **keydown** or **keyup** handler, all the keys that return zero above are indistinguishable. This bug was reported to Mozilla (bug 44259) in June 2000, and it took eight years to get the partial fixes out. Who knows when the rest (48434) will be fixed.

3.4. New Standard Key and Character Events

The DOM3 standard abandons all hope of creating order among `event.keyCode`, `event.which` and `event.charCode`, and instead defines new values for **keydown** and **keyup** events. For a while it deprecated the **keypress** event and replaced it with the **textInput** event, but that was undone. Only a few browsers implemented the first version, and, so far, no browsers have implemented the newest version.

Earlier versions of the specification defined attributes named `event.keyIdentifier` and `event.keyLocation`. The `keyIdentifier` was a string that in most cases looked like `"U+0041"` where the `"0041"` part is the unicode value of the character sent by the key when it is typed without modifiers, in this case the letter "A". For keys that didn't send unicode characters, or where the unicode value is not standardized, it was a string like "Enter", "Shift", "Left" or "F9". The `keyLocation` attribute gave values to distinguish among multiple keys that had the same identifier, like the left and right shift keys, or the keypad number keys. It was 0 for standard keys, 1 or 2 for left or right versions of a keys like Shift which appear twice on the keyboard, and 3 for keys on the numeric keypad.

WebKit implemented support for `keyIdentifier` and got it mostly right. Older versions conformed to an older version of the standard and returned two extra zeros (eg, `"U+000041"`) but this was corrected in version 525. Windows versions of Safari and Linux versions of Chrome return bad `keyIdentifier` values for all of the non-number symbol keys (WebKit Bug 19906 reported in July 2008). The `keyLocation` attribute is always 0 or 3, so it does not distinguish between left and right modifier keys.

Konqueror returns `keyIdentifier` values like "Shift" and "Enter" correctly, but instead of returning the Unicode values, it returns the typed character itself, `"a"` or `"A"` instead of `"U+0041"`. All `keyLocation` values are zero, except for modifiers key, which are always one, regardless of whether the left or right one was pressed.

We cannot, however expect any more browsers to implement that standard, since it has now changed. The DOM 3 standard no longer mentions `event.keyIdentifier` or `event.keyLocation`. Instead we have `event.key`, `event.char`, `event.location`.. So far as I know, no browser has yet implemented this new version of the DOM 3 standard.

In this standard `event.char` is defined only when you type a printable character, or another character with a defined code (like tab or backspace). It's basically like `event.charCode` except that it is the character, not the character code and can be any unicode character not just an ASCII code. `Event.key` is the same as `event.char` for printable keys. For other keys, even ones like tab or backspace that have character encodings, it is a string like `'Tab'`, `'Left'` or `'F9'`. These values are supposed to be the same on **keypress** events as they are on **keyup** and **keydown** events, though **keypress** would not be fired for those cases where `event.char` is null.

Note that neither of these pretends to be a keycode identifying a particular physical key on the keyboard. If you press the /? key on a US keyboard while shift is off, but press the shift key before releasing the /? key, then then on **keydown** you'll get `event.key=='/'` and on **keyup** you'll get `event.key=='?'`. The only way your Javascript program will know that those two events go together is if it happens to know that those two characters are on the same key. There is an `event.locale` value that is supposed to give you some clue on what type of keyboard is being used, but figuring out what keys go with what on a particular keyboard is up to you.

Clearly this abandonment of the idea of keycodes is going to cause problems, but is still probably justified. In many (most?) operating systems, I don't think the browser can actually tell which key was pressed. In the browser source code I've seen, the keycodes are generated from the the character codes, not vice versa, by simply assuming that the character came from a US keyboard. So the keycode values never really worked for non-US keyboards.

So while the keycode concept was a handly one, it isn't really practically extensible in the real world. If you want a keycode in the DOM 3 universe, you'll have to go on using the legacy `event.keyCode` value, which, standards or no standards, isn't going away. The DOM 3 standard seems to recognize this, and reluctantly provides an appendix with some standards for `event.keyCode` and the like. It casts a rather weak vote for what I called "IE keycodes" above.

4. Keyboard Focus

A computer has only one keyboard, but there are typically many things on the screen that could receive keyboard income. The place where keyboard input actually goes is said to have "keyboard focus".

Most of the time, browsers manage this sensibly for you, with fairly good consistency between browsers, and you don't have to worry about it. But in some applications you'll find your key event handlers failing to fire because the HTML element or window they are attached to have lost keyboard focus. This happens a lot when you have frames or iframes.

I haven't done enough study of this to write up a detailed discussion of keyboard focus, but if you need to actively manage keyboard focus, your main tools are the `focus()` and `blur()` methods that are defined for all HTML elements. There are also **focus** and **blur** events that are fired on elements when they gain or lose focus, to which you can attach event handlers.

Another tool useful in some situations is the HTML "tabindex" property, that defines the order in which keyboard focus moves through HTML objects as the user tabs through them. The element with tabindex equal to zero will be the one that starts with

keyboard focus when the page is loaded.

5. Conclusions

It's truely impressive what a hash has been made of a simple thing like recognizing a key pressed on the keyboard. You'd think computer technology would have advanced far enough by now to have this all worked out better.

The **keypress** events are generally the easiest to work with. They are likely to cause substantially fewer problems with non-US keyboard layouts and it's not too hard to identify which key was pressed. You can get the character typed by doing:

```
if (event.which == null)
   char= String.fromCharCode(event.keyCode);    // old IE
else if (event.which != 0 && event.charCode != 0)
   char= String.fromCharCode(event.which);      // All others
else
   // special key
```

What to do with **keypress** events on special keys is a problem. I recommend pretending they never happened. If you really want to process special key events, you should probably be working with **keydown** and **keyup** instead.

For **keydown** and **keyup** events, you can identify most common keys (letters, numbers, and a few others) by just looking at the `event.keyCode` and more or less pretending that it is an ASCII code. However, it isn't really, and the many Javascript manuals that say it can be converted to a character by doing "`String.fromCharCode(event.keyCode)`" are wrong. On **keydown** and **keyup** events, the keycodes are *not* character codes, and this conversion will give wild results for many keys. There is no general portable way to convert keycodes to characters. You pretty much have to sense the browser type and base the key mapping on that. I don't have information on keycodes sent by international keyboards.

Because of bugs, many keys cannot be distinguished on **keydown** and **keyup** in Macintosh Gecko.

Hope for sanity exists, with the new key event handling specifications in DOM3, but so far only WebKit implements them.

Last Update: Sun Nov 4 05:49:05 EST 2012