30th November 2019

Keyboard: keydown and keyup

Before we get to keyboard, please note that on modern devices there are other ways to “input something”. For instance, people use speech recognition (especially on mobile devices) or copy/paste with the mouse.

So if we want to track any input into an `<input>` field, then keyboard events are not enough. There’s another event named `input` to track changes of an `<input>` field, by any means. And it may be a better choice for such task. We’ll cover it later in the chapter Events: change, input, cut, copy, paste.

Keyboard events should be used when we want to handle keyboard actions (virtual keyboard also counts). For instance, to react on arrow keys `Up` and `Down` or hotkeys (including combinations of keys).

Teststand

To better understand keyboard events, you can use the teststand below.

Try different key combinations in the text field.

Result script.js style.css index.html

Prevent default for: ☐ keydown ☐ keyup

Ignore: ☐ keydown ☐ keyup

Focus on the input field and press a key.

Press keys here

Clear

Keydown and keyup

The `keydown` events happens when a key is pressed down, and then `keyup` – when it's released.

event.code and event.key

The `key` property of the event object allows to get the character, while the `code` property of the event object allows to get the "physical key code".

For instance, the same key `Z` can be pressed with or without `Shift`. That gives us two different characters: lowercase `z` and uppercase `Z`.

The `event.key` is exactly the character, and it will be different. But `event.code` is the same:

Key	event.key	event.code
`Z`	`z` (lowercase)	KeyZ
`Shift+Z`	`Z` (uppercase)	KeyZ

If a user works with different languages, then switching to another language would make a totally different character instead of `"Z"`. That will become the value of `event.key`, while `event.code` is always the same: `"KeyZ"`.

“KeyZ” and other key codes

Every key has the code that depends on its location on the keyboard. Key codes described in the UI Events code specification.

For instance:

- Letter keys have codes `"Key<letter>"`: `"KeyA"`, `"KeyB"` etc.
- Digit keys have codes: `"Digit<number>"`: `"Digit0"`, `"Digit1"` etc.
- Special keys are coded by their names: `"Enter"`, `"Backspace"`, `"Tab"` etc.

There are several widespread keyboard layouts, and the specification gives key codes for each of them.

Read the alphanumeric section of the spec for more codes, or just press a key in the teststand above.

Case matters: `"KeyZ"`, not `"keyZ"`

Seems obvious, but people still make mistakes.

Please evade mistypes: it's `KeyZ`, not `keyZ`. The check like `event.code=="keyZ"` won't work: the first letter of `"Key"` must be uppercase.

What if a key does not give any character? For instance, `Shift` or `F1` or others. For those keys, `event.key` is approximately the same as `event.code`:

Key	event.key	event.code
`F1`	F1	F1
`Backspace`	Backspace	Backspace
`Shift`	Shift	ShiftRight or ShiftLeft

Please note that `event.code` specifies exactly which key is pressed. For instance, most keyboards have two `Shift` keys: on the left and on the right side. The `event.code` tells us exactly which one was pressed, and `event.key` is responsible for the “meaning” of the key: what it is (a “Shift”).

Let's say, we want to handle a hotkey: `Ctrl+Z` (or `Cmd+Z` for Mac). Most text editors hook the “Undo” action on it. We can set a listener on `keydown` and check which key is pressed.

There's a dilemma here: in such a listener, should we check the value of `event.key` or `event.code`?

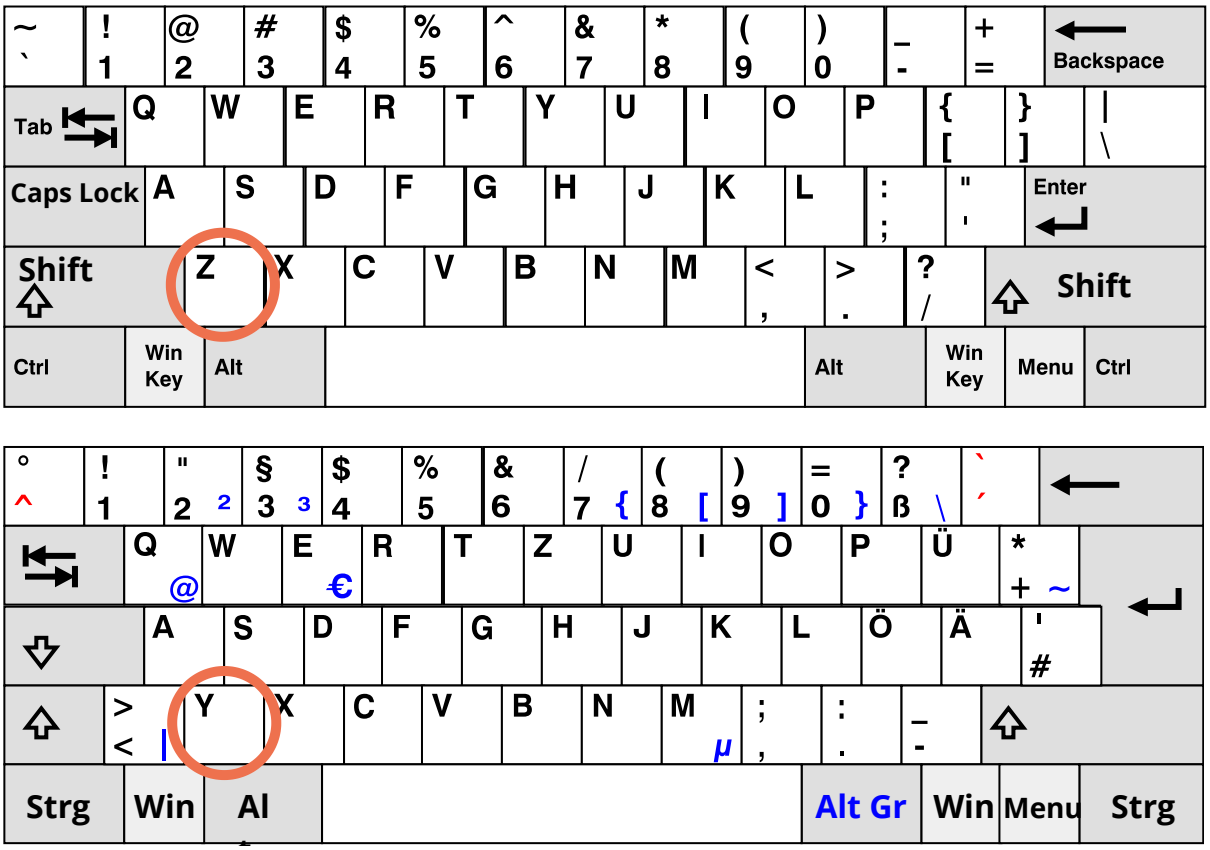
On one hand, the value of `event.key` is a character, it changes depending on the language. If the visitor has several languages in OS and switches between them, the same key gives different characters. So it makes sense to check `event.code`, it's always the same.

Like this:

```
1 document.addEventListener('keydown', function(event) {
2   if (event.code == 'KeyZ' && (event.ctrlKey || event.metaKey)) {
3     alert('Undo!')
4   }
5 });
```

On the other hand, there's a problem with `event.code`. For different keyboard layouts, the same key may have different characters.

For example, here are US layout ("QWERTY") and German layout ("QWERTZ") under it (from Wikipedia):



For the same key, US layout has "Z", while German layout has "Y" (letters are swapped).

Literally, `event.code` will equal `KeyZ` for people with German layout when they press `Y`.

If we check `event.code == 'KeyZ'` in our code, then for people with German layout such test will pass when they press `Y`.

That sounds really odd, but so it is. The specification explicitly mentions such behavior.

So, `event.code` may match a wrong character for unexpected layout. Same letters in different layouts may map to different physical keys, leading to different codes. Luckily, that happens only with several codes, e.g. `keyA`, `keyQ`, `keyZ` (as we've seen), and doesn't happen with special keys such as `Shift`. You can find the list in the specification.

To reliably track layout-dependent characters, `event.key` may be a better way.

On the other hand, `event.code` has the benefit of staying always the same, bound to the physical key location, even if the visitor changes languages. So hotkeys that rely on it work well even in case of a language switch.

Do we want to handle layout-dependant keys? Then `event.key` is the way to go.

Or we want a hotkey to work even after a language switch? Then `event.code` may be better.

Auto-repeat

If a key is being pressed for a long enough time, it starts to "auto-repeat": the `keydown` triggers again and again, and then when it's released we finally get `keyup`. So it's kind of normal to have many `keydown` and a

single `keyup`.

For events triggered by auto-repeat, the event object has `event.repeat` property set to `true`.

Default actions

Default actions vary, as there are many possible things that may be initiated by the keyboard.

For instance:

- A character appears on the screen (the most obvious outcome).
- A character is deleted (`Delete` key).
- The page is scrolled (`PageDown` key).
- The browser opens the “Save Page” dialog (`Ctrl+S`)
- …and so on.

Preventing the default action on `keydown` can cancel most of them, with the exception of OS-based special keys. For instance, on Windows `Alt+F4` closes the current browser window. And there’s no way to stop it by preventing the default action in JavaScript.

For instance, the `<input>` below expects a phone number, so it does not accept keys except digits, `+`, `()` or `-`:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')'
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Phone, please
```

Phone, please

Please note that special keys, such as `Backspace`, `Left`, `Right`, `Ctrl+V`, do not work in the input. That’s a side-effect of the strict filter `checkPhoneKey`.

Let’s relax it a little bit:

```
<script>
function checkPhoneKey(key) {
  return (key >= '0' && key <= '9') || key == '+' || key == '(' || key == ')'
    key == 'ArrowLeft' || key == 'ArrowRight' || key == 'Delete' || key == 'B
}
</script>
<input onkeydown="return checkPhoneKey(event.key)" placeholder="Phone, please
```

Phone, please

Now arrows and deletion works well.

…But we still can enter anything by using a mouse and right-click + Paste. So the filter is not 100% reliable. We can just let it be like that, because most of time it works. Or an alternative approach would be to track the `input` event – it triggers after any modification. There we can check the new value and highlight/modify it when it's invalid.

Legacy

In the past, there was a `keypress` event, and also `keyCode`, `charCode`, `which` properties of the event object.

There were so many browser incompatibilities while working with them, that developers of the specification had no way, other than deprecating all of them and creating new, modern events (described above in this chapter). The old code still works, as browsers keep supporting them, but there's totally no need to use those any more.

Summary

Pressing a key always generates a keyboard event, be it symbol keys or special keys like `Shift` or `Ctrl` and so on. The only exception is `Fn` key that sometimes presents on a laptop keyboard. There's no keyboard event for it, because it's often implemented on lower level than OS.

Keyboard events:

- `keydown` – on pressing the key (auto-repeats if the key is pressed for long),
- `keyup` – on releasing the key.

Main keyboard event properties:

- `code` – the "key code" (`"KeyA"`, `"ArrowLeft"` and so on), specific to the physical location of the key on keyboard.
- `key` – the character (`"A"`, `"a"` and so on), for non-character keys, such as `Esc`, usually has the same value as `code`.

In the past, keyboard events were sometimes used to track user input in form fields. That's not reliable, because the input can come from various sources. We have `input` and `change` events to handle any input (covered later in the chapter Events: change, input, cut, copy, paste). They trigger after any kind of input, including copy-pasting or speech recognition.

We should use keyboard events when we really want keyboard. For example, to react on hotkeys or special keys.

Tasks

Extended hotkeys

importance: 5

Create a function `runOnKeys(func, code1, code2, ... code_n)` that runs `func` on simultaneous pressing of keys with codes `code1`, `code2`, ..., `code_n`.

For instance, the code below shows `alert` when `"Q"` and `"W"` are pressed together (in any language, with or without CapsLock)

```
runOnKeys(
  () => alert("Hello!"),
  "KeyQ",
  "KeyW"
);
```

Demo in new window

solution

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)