

Part 3

Additional articles

JS

Ilya Kantor

Built at July 10, 2019

The last version of the tutorial is at <https://javascript.info>.

We constantly work to improve the tutorial. If you find any mistakes, please write at [our github](#).

- [Frames and windows](#)
 - [Popups and window methods](#)
 - [Cross-window communication](#)
 - [The clickjacking attack](#)
- [Binary data, files](#)
 - [ArrayBuffer, binary arrays](#)
 - [TextDecoder and TextEncoder](#)
 - [Blob](#)
 - [File and FileReader](#)
- [Network requests](#)
 - [Fetch](#)
 - [FormData](#)
 - [Fetch: Download progress](#)
 - [Fetch: Abort](#)
 - [Fetch: Cross-Origin Requests](#)
 - [Fetch API](#)
 - [URL objects](#)
 - [XMLHttpRequest](#)
 - [Resumable file upload](#)
 - [Long polling](#)
 - [WebSocket](#)
 - [Server Sent Events](#)
- [Storing data in the browser](#)
 - [Cookies, document.cookie](#)
 - [LocalStorage, sessionStorage](#)
 - [IndexedDB](#)
- [Animation](#)
 - [Bezier curve](#)
 - [CSS-animations](#)
 - [JavaScript animations](#)
- [Web components](#)
 - [From the orbital height](#)

- Custom elements
- Shadow DOM
- Template element
- Shadow DOM slots, composition
- Shadow DOM styling
- Shadow DOM and events
- Regular expressions
 - Patterns and flags
 - Methods of RegExp and String
 - Character classes
 - Escaping, special characters
 - Sets and ranges [...]
 - Quantifiers +, *, ? and {n}
 - Greedy and lazy quantifiers
 - Capturing groups
 - Backreferences in pattern: \n and \k
 - Alternation (OR) |
 - String start ^ and finish \$
 - Multiline mode, flag "m"
 - Lookahead and lookbehind
 - Infinite backtracking problem
 - Unicode: flag "u"
 - Unicode character properties \p
 - Sticky flag "y", searching at position

Frames and windows

Popups and window methods

A popup window is one of the oldest methods to show additional document to user. Basically, you just run:

```
window.open('https://javascript.info/')
```

...And it will open a new window with given URL. Most modern browsers are configured to open new tabs instead of separate windows.

Popups exist from really ancient times. The initial idea was to show another content without closing the main window. As of now, there are other ways to do that: we can load content dynamically with `fetch` and show it in a dynamically generated `<div>`. So, popups is not something we use everyday.

Also, popups are tricky on mobile devices, that don't show multiple windows simultaneously.

Still, there are tasks where popups are still used, e.g. for OAuth authorization (login with Google/Facebook/...), because:

1. A popup is a separate window with its own independent JavaScript environment. So opening a popup with a third-party non-trusted site is safe.
2. It's very easy to open a popup.
3. A popup can navigate (change URL) and send messages to the opener window.

Popup blocking

In the past, evil sites abused popups a lot. A bad page could open tons of popup windows with ads. So now most browsers try to block popups and protect the user.

Most browsers block popups if they are called outside of user-triggered event handlers like `onclick`.

For example:

```
// popup blocked
window.open('https://javascript.info');

// popup allowed
button.onclick = () => {
  window.open('https://javascript.info');
};
```

This way users are somewhat protected from unwanted popups, but the functionality is not disabled totally.

What if the popup opens from `onclick`, but after `setTimeout`? That's a bit tricky.

Try this code:

```
// open after 3 seconds
setTimeout(() => window.open('http://google.com'), 3000);
```

The popup opens in Chrome, but gets blocked in Firefox.

...If we decrease the delay, the popup works in Firefox too:

```
// open after 1 seconds
setTimeout(() => window.open('http://google.com'), 1000);
```

The difference is that Firefox treats a timeout of 2000ms or less as acceptable, but after it – removes the “trust”, assuming that now it’s “outside of the user action”. So the first one is blocked, and the second one is not.

window.open

The syntax to open a popup is: `window.open(url, name, params)`:

url

An URL to load into the new window.

name

A name of the new window. Each window has a `window.name`, and here we can specify which window to use for the popup. If there's already a window with such name – the given URL opens in it, otherwise a new window is opened.

params

The configuration string for the new window. It contains settings, delimited by a comma. There must be no spaces in params, for instance:

`width:200,height=100`.

Settings for `params`:

- Position:

- `left/top` (numeric) – coordinates of the window top-left corner on the screen. There is a limitation: a new window cannot be positioned offscreen.
- `width/height` (numeric) – width and height of a new window. There is a limit on minimal width/height, so it's impossible to create an invisible window.
- Window features:
 - `menubar` (yes/no) – shows or hides the browser menu on the new window.
 - `toolbar` (yes/no) – shows or hides the browser navigation bar (back, forward, reload etc) on the new window.
 - `location` (yes/no) – shows or hides the URL field in the new window. FF and IE don't allow to hide it by default.
 - `status` (yes/no) – shows or hides the status bar. Again, most browsers force it to show.
 - `resizable` (yes/no) – allows to disable the resize for the new window. Not recommended.
 - `scrollbars` (yes/no) – allows to disable the scrollbars for the new window. Not recommended.

There is also a number of less supported browser-specific features, which are usually not used. Check [window.open in MDN](#) ↗ for examples.

Example: a minimalistic window

Let's open a window with minimal set of features just to see which of them browser allows to disable:

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,width=0,height=0,left=-1000,top=-1000`;

open('/', 'test', params);
```

Here most “window features” are disabled and window is positioned offscreen. Run it and see what really happens. Most browsers “fix” odd things like zero `width/height` and offscreen `left/top`. For instance, Chrome open such a window with full width/height, so that it occupies the full screen.

Let's add normal positioning options and reasonable `width`, `height`, `left`, `top` coordinates:

```
let params = `scrollbars=no,resizable=no,status=no,location=no,toolbar=no,menubar=no,width=600,height=300,left=100,top=100`;

open('/', 'test', params);
```

Most browsers show the example above as required.

Rules for omitted settings:

- If there is no 3rd argument in the `open` call, or it is empty, then the default window parameters are used.
- If there is a string of params, but some `yes/no` features are omitted, then the omitted features assumed to have `no` value. So if you specify params, make sure you explicitly set all required features to yes.
- If there is no `left/top` in params, then the browser tries to open a new window near the last opened window.
- If there is no `width/height`, then the new window will be the same size as the last opened.

Accessing popup from window

The `open` call returns a reference to the new window. It can be used to manipulate it's properties, change location and even more.

In this example, we generate popup content from JavaScript:

```
let newWin = window.open("about:blank", "hello", "width=200,height=200");  
  
newWin.document.write("Hello, world!");
```

And here we modify the contents after loading:

```
let newWindow = open('/', 'example', 'width=300,height=300')  
newWindow.focus();  
  
alert(newWin.location.href); // (*) about:blank, loading hasn't started yet  
  
newWindow.onload = function() {  
  let html = `  newWindow.document.body.insertAdjacentHTML('afterbegin', html);  
};
```

Please note: immediately after `window.open`, the new window isn't loaded yet. That's demonstrated by `alert` in line `(*)`. So we wait for `onload` to modify it. We could also use `DOMContentLoaded` handler for `newWin.document`.

Same origin policy

Windows may freely access content of each other only if they come from the same origin (the same protocol://domain:port).

Otherwise, e.g. if the main window is from `site.com`, and the popup from `gmail.com`, that's impossible for user safety reasons. For the details, see chapter [Cross-window communication](#).

Accessing window from popup

A popup may access the “opener” window as well using `window.opener` reference. It is `null` for all windows except popups.

If you run the code below, it replaces the opener (current) window content with “Test”:

```
let newWin = window.open("about:blank", "hello", "width=200,height=200");

newWin.document.write(
  "<script>window.opener.document.body.innerHTML = 'Test'</script>"
);
```

So the connection between the windows is bidirectional: the main window and the popup have a reference to each other.

Closing a popup

To close a window: `win.close()`.

To check if a window is closed: `win.closed`.

Technically, the `close()` method is available for any `window`, but `window.close()` is ignored by most browsers if `window` is not created with `window.open()`. So it'll only work on a popup.

The `closed` property is `true` if the window is closed. That's useful to check if the popup (or the main window) is still open or not. A user can close it anytime, and our code should take that possibility into account.

This code loads and then closes the window:

```
let newWindow = open('/', 'example', 'width=300,height=300');

newWindow.onload = function() {
  newWindow.close();
};
```



```
alert(newWindow.closed); // true  
};
```

Scrolling and resizing

There are methods to move/resize a window:

`win.moveBy(x, y)`

Move the window relative to current position `x` pixels to the right and `y` pixels down. Negative values are allowed (to move left/up).

`win.moveTo(x, y)`

Move the window to coordinates `(x, y)` on the screen.

`win.resizeBy(width, height)`

Resize the window by given `width/height` relative to the current size. Negative values are allowed.

`win.resizeTo(width, height)`

Resize the window to the given size.

There's also `window.onresize` event.

Only popups

To prevent abuse, the browser usually blocks these methods. They only work reliably on popups that we opened, that have no additional tabs.

No minification/maximization

JavaScript has no way to minify or maximize a window. These OS-level functions are hidden from Frontend-developers.

Move/resize methods do not work for maximized/minimized windows.

Scrolling a window

We already talked about scrolling a window in the chapter [Window sizes and scrolling](#).

`win.scrollBy(x, y)`

Scroll the window `x` pixels right and `y` down relative the current scroll. Negative values are allowed.

`win.scrollTo(x,y)`

Scroll the window to the given coordinates `(x,y)`.

`elem.scrollToView(top = true)`

Scroll the window to make `elem` show up at the top (the default) or at the bottom for `elem.scrollToView(false)`.

There's also `window.onscroll` event.

Focus/blur on a window

Theoretically, there are `window.focus()` and `window.blur()` methods to focus/unfocus on a window. Also there are `focus/blur` events that allow to focus a window and catch the moment when the visitor switches elsewhere.

In the past evil pages abused those. For instance, look at this code:

```
window.onblur = () => window.focus();
```

When a user attempts to switch out of the window (`blur`), it brings it back to focus. The intention is to “lock” the user within the `window`.

So, there are limitations that forbid the code like that. There are many limitations to protect the user from ads and evils pages. They depend on the browser.

For instance, a mobile browser usually ignores that call completely. Also focusing doesn't work when a popup opens in a separate tab rather than a new window.

Still, there are some things that can be done.

For instance:

- When we open a popup, it's might be a good idea to run a `newWindow.focus()` on it. Just in case, for some OS/browser combinations it ensures that the user is in the new window now.
- If we want to track when a visitor actually uses our web-app, we can track `window.onfocus/onblur`. That allows us to suspend/resume in-page activities, animations etc. But please note that the `blur` event means that the visitor switched out from the window, but they still may observe it. The window is in the background, but still may be visible.

Summary

Popup windows are used rarely, as there are alternatives: loading and displaying information in-page, or in `iframe`.

If we're going to open a popup, a good practice is to inform the user about it. An "opening window" icon near a link or button would allow the visitor to survive the focus shift and keep both windows in mind.

- A popup can be opened by the `open(url, name, params)` call. It returns the reference to the newly opened window.
- Browsers block `open` calls from the code outside of user actions. Usually a notification appears, so that a user may allow them.
- Browsers open a new tab by default, but if sizes are provided, then it'll be a popup window.
- The popup may access the opener window using the `window.opener` property.
- The main window and the popup can freely read and modify each other if they have the same origin. Otherwise, they can change location of each other and [exchange messages].

To close the popup: use `close()` call. Also the user may close them (just like any other windows). The `window.closed` is `true` after that.

- Methods `focus()` and `blur()` allow to focus/unfocus a window. But they don't work all the time.
- Events `focus` and `blur` allow to track switching in and out of the window. But please note that a window may still be visible even in the background state, after `blur`.

Cross-window communication

The "Same Origin" (same site) policy limits access of windows and frames to each other.

The idea is that if a user has two pages open: one from `john-smith.com`, and another one is `gmail.com`, then they wouldn't want a script from `john-smith.com` to read our mail from `gmail.com`. So, the purpose of the "Same Origin" policy is to protect users from information theft.

Same Origin

Two URLs are said to have the "same origin" if they have the same protocol, domain and port.

These URLs all share the same origin:

- `http://site.com`
- `http://site.com/`
- `http://site.com/my/page.html`

These ones do not:

- `http://www.site.com` (another domain: `www.` matters)
- `http://site.org` (another domain: `.org` matters)
- `https://site.com` (another protocol: `https`)
- `http://site.com:8080` (another port: `8080`)

The “Same Origin” policy states that:

- if we have a reference to another window, e.g. a popup created by `window.open` or a window inside `<iframe>`, and that window comes from the same origin, then we have full access to that window.
- otherwise, if it comes from another origin, then we can't access the content of that window: variables, document, anything. The only exception is `location`: we can change it (thus redirecting the user). But we cannot *read* `location` (so we can't see where the user is now, no information leak).

In action: iframe

An `<iframe>` tag hosts a separate embedded window, with its own separate document and window objects.

We can access them using properties:

- `iframe.contentWindow` to get the window inside the `<iframe>`.
- `iframe.contentDocument` to get the document inside the `<iframe>`, короткий аналог `iframe.contentWindow.document`.

When we access something inside the embedded window, the browser checks if the iframe has the same origin. If that's not so then the access is denied (writing to `location` is an exception, it's still permitted).

For instance, let's try reading and writing to `<iframe>` from another origin:

```
<iframe src="https://example.com" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // we can get the reference to the inner window
    let iframeWindow = iframe.contentWindow; // OK
    try {
      // ...but not to the document inside it
      let doc = iframe.contentDocument; // ERROR
    } catch(e) {
      alert(e); // Security Error (another origin)
    }

    // also we can't READ the URL of the page in iframe
  }
}
```

```

try {
  // Can't read URL from the Location object
  let href = iframe.contentWindow.location.href; // ERROR
} catch(e) {
  alert(e); // Security Error
}

// ...we can WRITE into location (and thus load something else into the iframe)
iframe.contentWindow.location = '/'; // OK

iframe.onload = null; // clear the handler, not to run it after the location change
};
</script>

```

The code above shows errors for any operations except:

- Getting the reference to the inner window `iframe.contentWindow` – that's allowed.
- Writing to `location`.

Contrary to that, if the `<iframe>` has the same origin, we can do anything with it:

```

<!-- iframe from the same site -->
<iframe src="/" id="iframe"></iframe>

<script>
  iframe.onload = function() {
    // just do anything
    iframe.contentDocument.body.prepend("Hello, world!");
  };
</script>

```

`iframe.onload` vs `iframe.contentWindow.onload`

The `iframe.onload` event (on the `<iframe>` tag) is essentially the same as `iframe.contentWindow.onload` (on the embedded window object). It triggers when the embedded window fully loads with all resources.

...But we can't access `iframe.contentWindow.onload` for an iframe from another origin, so using `iframe.onload`.

Windows on subdomains: `document.domain`

By definition, two URLs with different domains have different origins.

But if windows share the same second-level domain, for instance `john.site.com`, `peter.site.com` and `site.com` (so that their common second-level domain is `site.com`), we can make the browser ignore that difference, so that they can be treated as coming from the “same origin” for the purposes of cross-window communication.

To make it work, each such window should run the code:

```
document.domain = 'site.com';
```

That's all. Now they can interact without limitations. Again, that's only possible for pages with the same second-level domain.

Iframe: wrong document pitfall

When an iframe comes from the same origin, and we may access its `document`, there's a pitfall. It's not related to cross-domain things, but important to know.

Upon its creation an iframe immediately has a document. But that document is different from the one that loads into it!

So if we do something with the document immediately, that will probably be lost.

Here, look:

```
<iframe src="/" id="iframe"></iframe>

<script>
  let oldDoc = iframe.contentDocument;
  iframe.onload = function() {
    let newDoc = iframe.contentDocument;
    // the loaded document is not the same as initial!
    alert(oldDoc == newDoc); // false
  };
</script>
```

We shouldn't work with the document of a not-yet-loaded iframe, because that's the *wrong document*. If we set any event handlers on it, they will be ignored.

How to detect the moment when the document is there?

The right document is definitely at place when `iframe.onload` triggers. But it only triggers when the whole iframe with all resources is loaded.

We can try to catch the moment earlier using checks in `setInterval`:

```

<iframe src="/" id="iframe"></iframe>

<script>
  let oldDoc = iframe.contentDocument;

  // every 100 ms check if the document is the new one
  let timer = setInterval(() => {
    let newDoc = iframe.contentDocument;
    if (newDoc == oldDoc) return;

    alert("New document is here!");

    clearInterval(timer); // cancel setInterval, don't need it any more
  }, 100);
</script>

```

Collection: window.frames

An alternative way to get a window object for `<iframe>` – is to get it from the named collection `window.frames`:

- By number: `window.frames[0]` – the window object for the first frame in the document.
- By name: `window.frames.iframeName` – the window object for the frame with `name="iframeName"`.

For instance:

```

<iframe src="/" style="height:80px" name="win" id="iframe"></iframe>

<script>
  alert(iframe.contentWindow == frames[0]); // true
  alert(iframe.contentWindow == frames.win); // true
</script>

```

An iframe may have other iframes inside. The corresponding `window` objects form a hierarchy.

Navigation links are:

- `window.frames` – the collection of “children” windows (for nested frames).
- `window.parent` – the reference to the “parent” (outer) window.
- `window.top` – the reference to the topmost parent window.

For instance:

```
window.frames[0].parent === window; // true
```

We can use the `top` property to check if the current document is open inside a frame or not:

```
if (window == top) { // current window == window.top?
  alert('The script is in the topmost window, not in a frame');
} else {
  alert('The script runs in a frame!');
}
```

The “sandbox” iframe attribute

The `sandbox` attribute allows for the exclusion of certain actions inside an `<iframe>` in order to prevent it executing untrusted code. It “sandboxes” the iframe by treating it as coming from another origin and/or applying other limitations.

There’s a “default set” of restrictions applied for `<iframe sandbox src="...">`. But it can be relaxed if we provide a space-separated list of restrictions that should not be applied as a value of the attribute, like this: `<iframe sandbox="allow-forms allow-popups">`.

In other words, an empty `"sandbox"` attribute puts the strictest limitations possible, but we can put a space-delimited list of those that we want to lift.

Here’s a list of limitations:

allow-same-origin

By default `"sandbox"` forces the “different origin” policy for the iframe. In other words, it makes the browser to treat the `iframe` as coming from another origin, even if its `src` points to the same site. With all implied restrictions for scripts. This option removes that feature.

allow-top-navigation

Allows the `iframe` to change `parent.location`.

allow-forms

Allows to submit forms from `iframe`.

allow-scripts

Allows to run scripts from the `iframe`.

allow-popups

Allows to `window.open` popups from the `iframe`

See [the manual](#) for more.

The example below demonstrates a sandboxed `iframe` with the default set of restrictions: `<iframe sandbox src="...">`. It has some JavaScript and a form.

Please note that nothing works. So the default set is really harsh:

<https://plnkr.co/edit/GAhzx0j3JwAB1TMzwyxL?p=preview>

i Please note:

The purpose of the `"sandbox"` attribute is only to *add more* restrictions. It cannot remove them. In particular, it can't relax same-origin restrictions if the `iframe` comes from another origin.

Cross-window messaging

The `postMessage` interface allows windows to talk to each other no matter which origin they are from.

So, it's a way around the "Same Origin" policy. It allows a window from `john-smith.com` to talk to `gmail.com` and exchange information, but only if they both agree and call corresponding JavaScript functions. That makes it safe for users.

The interface has two parts.

postMessage

The window that wants to send a message calls `postMessage` method of the receiving window. In other words, if we want to send the message to `win`, we should call `win.postMessage(data, targetOrigin)`.

Arguments:

data

The data to send. Can be any object, the data is cloned using the "structured cloning algorithm". IE supports only strings, so we should `JSON.stringify` complex objects to support that browser.

targetOrigin

Specifies the origin for the target window, so that only a window from the given origin will get the message.

The `targetOrigin` is a safety measure. Remember, if the target window comes from another origin, we can't read its `location` in the sender window. So we can't

be sure which site is open in the intended window right now: the user could navigate away, and the sender window has no idea about it.

Specifying `targetOrigin` ensures that the window only receives the data if it's still at the right site. Important when the data is sensitive.

For instance, here `win` will only receive the message if it has a document from the origin `http://example.com`:

```
<iframe src="http://example.com" name="example">

<script>
  let win = window.frames.example;

  win.postMessage("message", "http://example.com");
</script>
```

If we don't want that check, we can set `targetOrigin` to `*`.

```
<iframe src="http://example.com" name="example">

<script>
  let win = window.frames.example;

  win.postMessage("message", "*");
</script>
```

onmessage

To receive a message, the target window should have a handler on the `message` event. It triggers when `postMessage` is called (and `targetOrigin` check is successful).

The event object has special properties:

data

The data from `postMessage`.

origin

The origin of the sender, for instance `http://javascript.info`.

source

The reference to the sender window. We can immediately `source.postMessage(...)` back if we want.

To assign that handler, we should use `addEventListener`, a short syntax `window.onmessage` does not work.

Here's an example:

```
window.addEventListener("message", function(event) {  
  if (event.origin !== 'http://javascript.info') {  
    // something from an unknown domain, let's ignore it  
    return;  
  }  
  
  alert( "received: " + event.data );  
  
  // can message back using event.source.postMessage(...)  
});
```

The full example:

<https://plnkr.co/edit/ltrzlGvN8UPdpMtyxll9?p=preview> ↗

There's no delay

There's totally no delay between `postMessage` and the `message` event. The event triggers synchronously, faster than `setTimeout(..., 0)`.

Summary

To call methods and access the content of another window, we should first have a reference to it.

For popups we have these references:

- From the opener window: `window.open` – opens a new window and returns a reference to it,
- From the popup: `window.opener` – is a reference to the opener window from a popup.

For iframes, we can access parent/children windows using:

- `window.frames` – a collection of nested window objects,
- `window.parent`, `window.top` are the references to parent and top windows,
- `iframe.contentWindow` is the window inside an `<iframe>` tag.

If windows share the same origin (host, port, protocol), then windows can do whatever they want with each other.

Otherwise, only possible actions are:

- Change the `location` of another window (write-only access).
- Post a message to it.

Exceptions are:

- Windows that share the same second-level domain: `a.site.com` and `b.site.com`. Then setting `document.domain='site.com'` in both of them puts them into the “same origin” state.
- If an `iframe` has a `sandbox` attribute, it is forcefully put into the “different origin” state, unless the `allow-same-origin` is specified in the attribute value. That can be used to run untrusted code in iframes from the same site.

The `postMessage` interface allows two windows with any origins to talk:

1. The sender calls `targetWin.postMessage(data, targetOrigin)`.
2. If `targetOrigin` is not `'*'`, then the browser checks if window `targetWin` has the origin `targetOrigin`.
3. If it is so, then `targetWin` triggers the `message` event with special properties:
 - `origin` – the origin of the sender window (like `http://my.site.com`)
 - `source` – the reference to the sender window.
 - `data` – the data, any object in everywhere except IE that supports only strings.

We should use `addEventListener` to set the handler for this event inside the target window.

The clickjacking attack

The “clickjacking” attack allows an evil page to click on a “victim site” *on behalf of the visitor*.

Many sites were hacked this way, including Twitter, Facebook, Paypal and other sites. They have all been fixed, of course.

The idea

The idea is very simple.

Here’s how clickjacking was done with Facebook:

1. A visitor is lured to the evil page. It doesn’t matter how.

2. The page has a harmless-looking link on it (like “get rich now” or “click here, very funny”).
3. Over that link the evil page positions a transparent `<iframe>` with `src` from facebook.com, in such a way that the “Like” button is right above that link. Usually that’s done with `z-index`.
4. In attempting to click the link, the visitor in fact clicks the button.

The demo

Here’s how the evil page looks. To make things clear, the `<iframe>` is half-transparent (in real evil pages it’s fully transparent):

```
<style>
iframe { /* iframe from the victim site */
  width: 400px;
  height: 100px;
  position: absolute;
  top:0; left:-20px;
  opacity: 0.5; /* in real opacity:0 */
  z-index: 1;
}
</style>

<div>Click to get rich now:</div>

<!-- The url from the victim site -->
<iframe src="/clickjacking/facebook.html"></iframe>

<button>Click here!</button>

<div>...And you're cool (I'm a cool hacker actually)!</div>
```

The full demo of the attack:

<https://plnkr.co/edit/GQKK8Zc7DXT3KdV7tQUu?p=preview> ↗

Here we have a half-transparent `<iframe src="facebook.html">`, and in the example we can see it hovering over the button. A click on the button actually clicks on the iframe, but that’s not visible to the user, because the iframe is transparent.

As a result, if the visitor is authorized on Facebook (“remember me” is usually turned on), then it adds a “Like”. On Twitter that would be a “Follow” button.

Here’s the same example, but closer to reality, with `opacity:0` for `<iframe>`:

<https://plnkr.co/edit/aebDnhU3B7c6d2QN5Rhy?p=preview> ↗

All we need to attack – is to position the `<iframe>` on the evil page in such a way that the button is right over the link. So that when a user clicks the link, they actually click the button. That's usually doable with CSS.

i Clickjacking is for clicks, not for keyboard

The attack only affects mouse actions (or similar, like taps on mobile).

Keyboard input is much difficult to redirect. Technically, if we have a text field to hack, then we can position an iframe in such a way that text fields overlap each other. So when a visitor tries to focus on the input they see on the page, they actually focus on the input inside the iframe.

But then there's a problem. Everything that the visitor types will be hidden, because the iframe is not visible.

People will usually stop typing when they can't see their new characters printing on the screen.

Old-school defences (weak)

The oldest defence is a bit of JavaScript which forbids opening the page in a frame (so-called "framebusting").

That looks like this:

```
if (top !== window) {  
    top.location = window.location;  
}
```

That is: if the window finds out that it's not on top, then it automatically makes itself the top.

This not a reliable defence, because there are many ways to hack around it. Let's cover a few.

Blocking top-navigation

We can block the transition caused by changing `top.location` in `beforeunload` event handler.

The top page (enclosing one, belonging to the hacker) sets a preventing handler to it, like this:

```
window.onbeforeunload = function() {  
    return false;  
};
```

When the `iframe` tries to change `top.location`, the visitor gets a message asking them whether they want to leave.

In most cases the visitor would answer negatively because they don't know about the `iframe` – all they can see is the top page, there's no reason to leave. So `top.location` won't change!

In action:

<https://plnkr.co/edit/WCEMuiV3PmW1klyyf6FH?p=preview> ↗

Sandbox attribute

One of the things restricted by the `sandbox` attribute is navigation. A sandboxed `iframe` may not change `top.location`.

So we can add the `iframe` with `sandbox="allow-scripts allow-forms"`. That would relax the restrictions, permitting scripts and forms. But we omit `allow-top-navigation` so that changing `top.location` is forbidden.

Here's the code:

```
<iframe sandbox="allow-scripts allow-forms" src="facebook.html"></iframe>
```

There are other ways to work around that simple protection too.

X-Frame-Options

The server-side header `X-Frame-Options` can permit or forbid displaying the page inside a frame.

It must be sent exactly as HTTP-header: the browser will ignore it if found in HTML `<meta>` tag. So, `<meta http-equiv="X-Frame-Options" ...>` won't do anything.

The header may have 3 values:

DENY

Never ever show the page inside a frame.

SAMEORIGIN

Allow inside a frame if the parent document comes from the same origin.

ALLOW-FROM domain

Allow inside a frame if the parent document is from the given domain.

For instance, Twitter uses `X-Frame-Options: SAMEORIGIN`.

Showing with disabled functionality

The `X-Frame-Options` header has a side-effect. Other sites won't be able to show our page in a frame, even if they have good reasons to do so.

So there are other solutions... For instance, we can “cover” the page with a `<div>` with styles `height: 100%; width: 100%;`, so that it will intercept all clicks. That `<div>` is to be removed if `window == top` or if we figure out that we don't need the protection.

Something like this:

```
<style>
  #protector {
    height: 100%;
    width: 100%;
    position: absolute;
    left: 0;
    top: 0;
    z-index: 9999999;
  }
</style>

<div id="protector">
  <a href="/" target="_blank">Go to the site</a>
</div>

<script>
  // there will be an error if top window is from the different origin
  // but that's ok here
  if (top.document.domain !== document.domain) {
    protector.remove();
  }
</script>
```

The demo:

<https://plnkr.co/edit/WuzXGKQamIVp1sE08svn?p=preview> ↗

Samesite cookie attribute

The `samesite` cookie attribute can also prevent clickjacking attacks.

A cookie with such attribute is only sent to a website if it's opened directly, not via a frame, or otherwise. More information in the chapter [Cookies, document.cookie](#).

If the site, such as Facebook, had `samesite` attribute on its authentication cookie, like this:


```
Set-Cookie: authorization=secret; samesite
```

...Then such cookie wouldn't be sent when Facebook is open in iframe from another site. So the attack would fail.

The `samesite` cookie attribute will not have an effect when cookies are not used. This may allow other websites to easily show our public, unauthenticated pages in iframes.

However, this may also allow clickjacking attacks to work in a few limited cases. An anonymous polling website that prevents duplicate voting by checking IP addresses, for example, would still be vulnerable to clickjacking because it does not authenticate users using cookies.

Summary

Clickjacking is a way to “trick” users into clicking on a victim site without even knowing what's happening. That's dangerous if there are important click-activated actions.

A hacker can post a link to their evil page in a message, or lure visitors to their page by some other means. There are many variations.

From one perspective – the attack is “not deep”: all a hacker is doing is intercepting a single click. But from another perspective, if the hacker knows that after the click another control will appear, then they may use cunning messages to coerce the user into clicking on them as well.

The attack is quite dangerous, because when we engineer the UI we usually don't anticipate that a hacker may click on behalf of the visitor. So vulnerabilities can be found in totally unexpected places.

- It is recommended to use `X-Frame-Options: SAMEORIGIN` on pages (or whole websites) which are not intended to be viewed inside frames.
- Use a covering `<div>` if we want to allow our pages to be shown in iframes, but still stay safe.

Binary data, files

Working with binary data and files in JavaScript.

ArrayBuffer, binary arrays

In web-development we meet binary data mostly while dealing with files (create, upload, download). Another typical use case is image processing.

That's all possible in JavaScript, and binary operations are high-performant.

Although, there's a bit of confusion, because there are many classes. To name a few:

- `ArrayBuffer`, `Uint8Array`, `DataView`, `Blob`, `File`, etc.

Binary data in JavaScript is implemented in a non-standard way, compared to other languages. But when we sort things out, everything becomes fairly simple.

The basic binary object is `ArrayBuffer` – a reference to a fixed-length contiguous memory area.

We create it like this:

```
let buffer = new ArrayBuffer(16); // create a buffer of length 16
alert(buffer.byteLength); // 16
```

This allocates a contiguous memory area of 16 bytes and pre-fills it with zeroes.

`ArrayBuffer` is not an array of something

Let's eliminate a possible source of confusion. `ArrayBuffer` has nothing in common with `Array`:

- It has a fixed length, we can't increase or decrease it.
- It takes exactly that much space in the memory.
- To access individual bytes, another "view" object is needed, not `buffer[index]`.

`ArrayBuffer` is a memory area. What's stored in it? It has no clue. Just a raw sequence of bytes.

To manipulate an `ArrayBuffer`, we need to use a "view" object.

A view object does not store anything on its own. It's the "eyeglasses" that give an interpretation of the bytes stored in the `ArrayBuffer`.

For instance:

- **`Uint8Array`** – treats each byte in `ArrayBuffer` as a separate number, with possible values are from 0 to 255 (a byte is 8-bit, so it can hold only that much). Such value is called a "8-bit unsigned integer".
- **`Uint16Array`** – treats every 2 bytes as an integer, with possible values from 0 to 65535. That's called a "16-bit unsigned integer".

- **Uint32Array** – treats every 4 bytes as an integer, with possible values from 0 to 4294967295. That’s called a “32-bit unsigned integer”.
- **Float64Array** – treats every 8 bytes as a floating point number with possible values from 5.0×10^{-324} to 1.8×10^{308} .

So, the binary data in an **ArrayBuffer** of 16 bytes can be interpreted as 16 “tiny numbers”, or 8 bigger numbers (2 bytes each), or 4 even bigger (4 bytes each), or 2 floating-point values with high precision (8 bytes each).

new ArrayBuffer(16)																
Uint8Array	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Uint16Array	0		1		2		3		4		5		6		7	
Uint32Array	0				1				2				3			
Float64Array	0								1							

ArrayBuffer is the core object, the root of everything, the raw binary data.

But if we’re going to write into it, or iterate over it, basically for almost any operation – we must use a view, e.g:

```
let buffer = new ArrayBuffer(16); // create a buffer of length 16

let view = new Uint32Array(buffer); // treat buffer as a sequence of 32-bit integers

alert(Uint32Array.BYTES_PER_ELEMENT); // 4 bytes per integer

alert(view.length); // 4, it stores that many integers
alert(view.byteLength); // 16, the size in bytes

// let's write a value
view[0] = 123456;

// iterate over values
for(let num of view) {
  alert(num); // 123456, then 0, 0, 0 (4 values total)
}
```

TypedArray

The common term for all these views (**Uint8Array**, **Uint32Array**, etc) is **TypedArray** [↗](#). They share the same set of methods and properties.

They are much more like regular arrays: have indexes and iterable.

A typed array constructor (be it `Int8Array` or `Float64Array`, doesn't matter) behaves differently depending on argument types.

There are 5 variants of arguments:

```
new TypedArray(buffer, [byteOffset], [length]);
new TypedArray(object);
new TypedArray(typedArray);
new TypedArray(length);
new TypedArray();
```

1. If an `ArrayBuffer` argument is supplied, the view is created over it. We used that syntax already.

Optionally we can provide `byteOffset` to start from (0 by default) and the `length` (till the end of the buffer by default), then the view will cover only a part of the `buffer`.

2. If an `Array`, or any array-like object is given, it creates a typed array of the same length and copies the content.

We can use it to pre-fill the array with the data:

```
let arr = new Uint8Array([0, 1, 2, 3]);
alert( arr.length ); // 4, created binary array of the same length
alert( arr[1] ); // 1, filled with 4 bytes (unsigned 8-bit integers) with given v
```

3. If another `TypedArray` is supplied, it does the same: creates a typed array of the same length and copies values. Values are converted to the new type in the process, if needed.

```
let arr16 = new Uint16Array([1, 1000]);
let arr8 = new Uint8Array(arr16);
alert( arr8[0] ); // 1
alert( arr8[1] ); // 232, tried to copy 1000, but can't fit 1000 into 8 bits (exp
```

4. For a numeric argument `length` – creates the typed array to contain that many elements. Its byte length will be `length` multiplied by the number of bytes in a single item `TypedArray.BYTES_PER_ELEMENT`:

```
let arr = new Uint16Array(4); // create typed array for 4 integers
alert( Uint16Array.BYTES_PER_ELEMENT ); // 2 bytes per integer
```

```
alert( arr.byteLength ); // 8 (size in bytes)
```

5. Without arguments, creates an zero-length typed array.

We can create a `TypedArray` directly, without mentioning `ArrayBuffer`. But a view cannot exist without an underlying `ArrayBuffer`, so gets created automatically in all these cases except the first one (when provided).

To access the `ArrayBuffer`, there are properties:

- `arr.buffer` – references the `ArrayBuffer`.
- `arr.byteLength` – the length of the `ArrayBuffer`.

So, we can always move from one view to another:

```
let arr8 = new Uint8Array([0, 1, 2, 3]);

// another view on the same data
let arr16 = new Uint16Array(arr8.buffer);
```

Here's the list of typed arrays:

- `Uint8Array`, `Uint16Array`, `Uint32Array` – for integer numbers of 8, 16 and 32 bits.
 - `Uint8ClampedArray` – for 8-bit integers, “clamps” them on assignment (see below).
- `Int8Array`, `Int16Array`, `Int32Array` – for signed integer numbers (can be negative).
- `Float32Array`, `Float64Array` – for signed floating-point numbers of 32 and 64 bits.

No `int8` or similar single-valued types

Please note, despite of the names like `Int8Array`, there's no single-value type like `int`, or `int8` in JavaScript.

That's logical, as `Int8Array` is not an array of these individual values, but rather a view on `ArrayBuffer`.

Out-of-bounds behavior

What if we attempt to write an out-of-bounds value into a typed array? There will be no error. But extra bits are cut-off.

For instance, let's try to put 256 into `Uint8Array`. In binary form, 256 is `100000000` (9 bits), but `Uint8Array` only provides 8 bits per value, that makes the available range from 0 to 255.

For bigger numbers, only the rightmost (less significant) 8 bits are stored, and the rest is cut off:

8-bit integer
1 `00000000` 256

So we'll get zero.

For 257, the binary form is `100000001` (9 bits), the rightmost 8 get stored, so we'll have `1` in the array:

8-bit integer
1 `00000001` 257

In other words, the number modulo 2^8 is saved.

Here's the demo:

```
let uint8array = new Uint8Array(16);

let num = 256;
alert(num.toString(2)); // 100000000 (binary representation)

uint8array[0] = 256;
uint8array[1] = 257;

alert(uint8array[0]); // 0
alert(uint8array[1]); // 1
```

`Uint8ClampedArray` is special in this aspect, its behavior is different. It saves 255 for any number that is greater than 255, and 0 for any negative number. That behavior is useful for image processing.

TypedArray methods

`TypedArray` has regular `Array` methods, with notable exceptions.

We can iterate, `map`, `slice`, `find`, `reduce` etc.

There are few things we can't do though:

- No `splice` – we can't “delete” a value, because typed arrays are views on a buffer, and these are fixed, contiguous areas of memory. All we can do is to

assign a zero.

- No `concat` method.

There are two additional methods:

- `arr.set(fromArr, [offset])` copies all elements from `fromArr` to the `arr`, starting at position `offset` (0 by default).
- `arr.subarray([begin, end])` creates a new view of the same type from `begin` to `end` (exclusive). That's similar to `slice` method (that's also supported), but doesn't copy anything – just creates a new view, to operate on the given piece of data.

These methods allow us to copy typed arrays, mix them, create new arrays from existing ones, and so on.

DataView

[DataView](#) is a special super-flexible “untyped” view over `ArrayBuffer`. It allows to access the data on any offset in any format.

- For typed arrays, the constructor dictates what the format is. The whole array is supposed to be uniform. The *i*-th number is `arr[i]`.
- With `DataView` we access the data with methods like `.getUint8(i)` or `.getUint16(i)`. We choose the format at method call time instead of the construction time.

The syntax:

```
new DataView(buffer, [byteOffset], [byteLength])
```

- **buffer** – the underlying `ArrayBuffer`. Unlike typed arrays, `DataView` doesn't create a buffer on its own. We need to have it ready.
- **byteOffset** – the starting byte position of the view (by default 0).
- **byteLength** – the byte length of the view (by default till the end of `buffer`).

For instance, here we extract numbers in different formats from the same buffer:

```
// binary array of 4 bytes, all have the maximal value 255
let buffer = new Uint8Array([255, 255, 255, 255]).buffer;

let dataView = new DataView(buffer);

// get 8-bit number at offset 0
```

```

alert( dataView.getUint8(0) ); // 255

// now get 16-bit number at offset 0, it consists of 2 bytes, together interpreted as
alert( dataView.getUint16(0) ); // 65535 (biggest 16-bit unsigned int)

// get 32-bit number at offset 0
alert( dataView.getUint32(0) ); // 4294967295 (biggest 32-bit unsigned int)

dataView.setUint32(0, 0); // set 4-byte number to zero, thus setting all bytes to 0

```

`DataView` is great when we store mixed-format data in the same buffer. E.g we store a sequence of pairs (16-bit integer, 32-bit float). Then `DataView` allows to access them easily.

Summary

`ArrayBuffer` is the core object, a reference to the fixed-length contiguous memory area.

To do almost any operation on `ArrayBuffer`, we need a view.

- It can be a `TypedArray`:
 - `Uint8Array`, `Uint16Array`, `Uint32Array` – for unsigned integers of 8, 16, and 32 bits.
 - `Uint8ClampedArray` – for 8-bit integers, “clamps” them on assignment.
 - `Int8Array`, `Int16Array`, `Int32Array` – for signed integer numbers (can be negative).
 - `Float32Array`, `Float64Array` – for signed floating-point numbers of 32 and 64 bits.
- Or a `DataView` – the view that uses methods to specify a format, e.g. `getUint8(offset)`.

In most cases we create and operate directly on typed arrays, leaving `ArrayBuffer` under cover, as a “common discriminator”. We can access it as `.buffer` and make another view if needed.

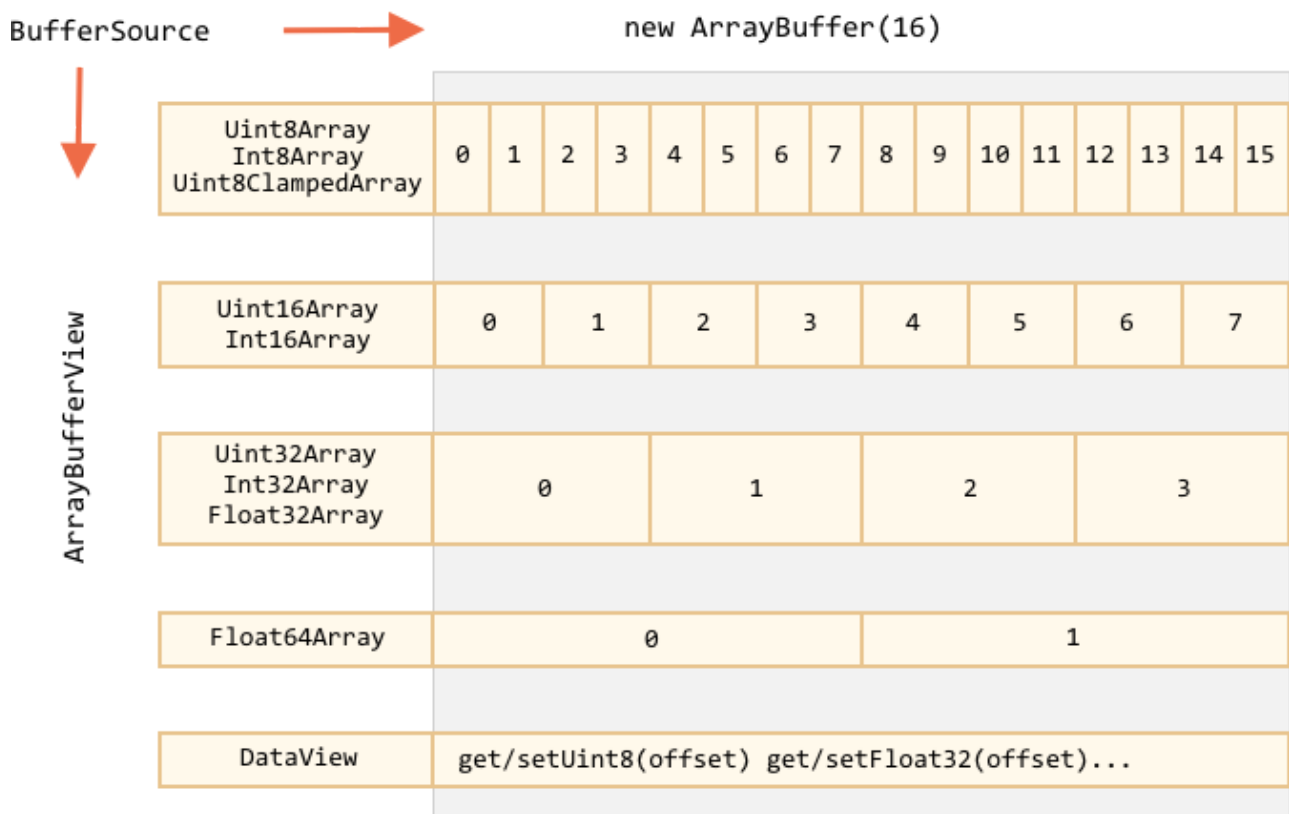
There are also two additional terms, that are used in descriptions of methods that operate on binary data:

- `ArrayBufferView` is an umbrella term for all these kinds of views.
- `BufferSource` is an umbrella term for `ArrayBuffer` or `ArrayBufferView`.

We'll see these terms in the next chapters. `BufferSource` is one of the most common terms, as it means “any kind of binary data” – an `ArrayBuffer` or a view

over it.

Here's a cheatsheet:



✓ Tasks

Concatenate typed arrays

Given an array of `Uint8Array`, write a function `concat(arrays)` that returns a concatenation of them into a single array.

[Open a sandbox with tests.](#)

[To solution](#)

TextDecoder and TextEncoder

What if the binary data is actually a string? For instance, we received a file with textual data.

The build-in `TextDecoder` object allows to read the value into an actual JavaScript string, given the buffer and the encoding.

We first need to create it:

```
let decoder = new TextDecoder([label], [options]);
```

- **label** – the encoding, `utf-8` by default, but `big5`, `windows-1251` and many other are also supported.
- **options** – optional object:
 - **fatal** – boolean, if `true` then throw an exception for invalid (non-decodable) characters, otherwise (default) replace them with character `\uFFFD`.
 - **ignoreBOM** – boolean, if `true` then ignore BOM (an optional byte-order unicode mark), rarely needed.

...And then decode:

```
let str = decoder.decode([input], [options]);
```

- **input** – `BufferSource` to decode.
- **options** – optional object:
 - **stream** – true for decoding streams, when `decoder` is called repeatedly with incoming chunks of data. In that case a multi-byte character may occasionally split between chunks. This options tells `TextDecoder` to memorize “unfinished” characters and decode them when the next chunk comes.

For instance:

```
let uint8Array = new Uint8Array([72, 101, 108, 108, 111]);  
  
alert( new TextDecoder().decode(uint8Array) ); // Hello
```

```
let uint8Array = new Uint8Array([228, 189, 160, 229, 165, 189]);  
  
alert( new TextDecoder().decode(uint8Array) ); // 你好
```


We can decode a part of the buffer by creating a subarray view for it:

```
let uint8Array = new Uint8Array([0, 72, 101, 108, 108, 111, 0]);  
  
// the string is in the middle  
// create a new view over it, without copying anything
```

```
let binaryString = uint8Array.subarray(1, -1);

alert( new TextDecoder().decode(binaryString) ); // Hello
```

TextEncoder

[TextEncoder](#)  does the reverse thing – converts a string into bytes.

The syntax is:

```
let encoder = new TextEncoder();
```

The only encoding it supports is “utf-8”.

It has two methods:


- **encode(str)** – returns `Uint8Array` from a string.
- **encodeInto(str, destination)** – encodes `str` into `destination` that must be `Uint8Array`.

```
let encoder = new TextEncoder();

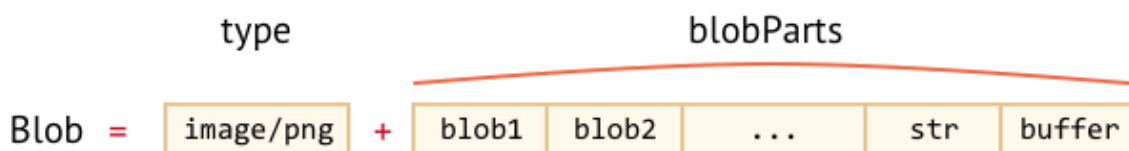
let uint8Array = encoder.encode("Hello");
alert(uint8Array); // 72,101,108,108,111
```

Blob

`ArrayBuffer` and views are a part of ECMA standard, a part of JavaScript.

In the browser, there are additional higher-level objects, described in [File API](#) , in particular `Blob`.

`Blob` consists of an optional string `type` (a MIME-type usually), plus `blobParts` – a sequence of other `Blob` objects, strings and `BufferSources`.



The constructor syntax is:

```
new Blob(blobParts, options);
```

- **blobParts** is an array of `Blob` / `BufferSource` / `String` values.
- **options** optional object:
 - **type** – blob type, usually MIME-type, e.g. `image/png`,
 - **endings** – whether to transform end-of-line to make the blob correspond to current OS newlines (`\r\n` or `\n`). By default `"transparent"` (do nothing), but also can be `"native"` (transform).

For example:

```
// create Blob from a string
let blob = new Blob(["<html>...</html>"], {type: 'text/html'});
// please note: the first argument must be an array [...]
```

```
// create Blob from a typed array and strings
let hello = new Uint8Array([72, 101, 108, 108, 111]); // "hello" in binary form

let blob = new Blob([hello, ' ', 'world'], {type: 'text/plain'});
```

We can extract blob slices with:

```
blob.slice([byteStart], [byteEnd], [contentType]);
```

- **byteStart** – the starting byte, by default 0.
- **byteEnd** – the last byte (exclusive, by default till the end).
- **contentType** – the `type` of the new blob, by default the same as the source.

The arguments are similar to `array.slice`, negative numbers are allowed too.

Blobs are immutable

We can't change data directly in a blob, but we can slice parts of blobs, create new blobs from them, mix them into a new blob and so on.

This behavior is similar to JavaScript strings: we can't change a character in a string, but we can make a new corrected string.

Blob as URL

A Blob can be easily used as an URL for `<a>`, `` or other tags, to show its contents.

Thanks to `type`, we can also download/upload blobs, and it naturally becomes `Content-Type` in network requests.

Let's start with a simple example. By clicking on a link you download a dynamically-generated blob with `hello world` contents as a file:

```
<!-- download attribute forces the browser to download instead of navigating -->
<a download="hello.txt" href="#" id="link">Download</a>

<script>
let blob = new Blob(["Hello, world!"], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);
</script>
```

We can also create a link dynamically in JavaScript and simulate a click by `link.click()`, then download starts automatically.

Here's the similar code that causes user to download the dynamically created Blob, without any HTML:

```
let link = document.createElement('a');
link.download = 'hello.txt';

let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

link.href = URL.createObjectURL(blob);

link.click();

URL.revokeObjectURL(link.href);
```

`URL.createObjectURL` takes a blob and creates a unique URL for it, in the form `blob:<origin>/<uuid>`.

That's what the value of `link.href` looks like:

```
blob:https://javascript.info/1e67e00e-860d-40a5-89ae-6ab0cbee6273
```

The browser for each url generated by `URL.createObjectURL` stores an the url → blob mapping internally. So such urls are short, but allow to access the blob.

A generated url (and hence the link with it) is only valid within the current document, while it's open. And it allows to reference the blob in ``, `<a>`, basically any other object that expects an url.

There's a side-effect though. While there's an mapping for a blob, the blob itself resides in the memory. The browser can't free it.

The mapping is automatically cleared on document unload, so blobs are freed then. But if an app is long-living, then that doesn't happen soon.

So if we create an URL, that blob will hang in memory, even if not needed any more.

`URL.revokeObjectURL(url)` removes the reference from the internal mapping, thus allowing the blob to be deleted (if there are no other references), and the memory to be freed.

In the last example, we intend the blob to be used only once, for instant downloading, so we call `URL.revokeObjectURL(link.href)` immediately.

In the previous example though, with the clickable HTML-link, we don't call `URL.revokeObjectURL(link.href)`, because that would make the blob url invalid. After the revocation, as the mapping is removed, the url doesn't work any more.

Blob to base64

An alternative to `URL.createObjectURL` is to convert a blob into a base64-encoded string.

That encoding represents binary data as a string of ultra-safe “readable” characters with ASCII-codes from 0 to 64. And what's more important – we can use this encoding in “data-urls”.

A [data url](#) has the form `data:[<mediatype>][;<base64>],<data>`. We can use such urls everywhere, on a par with “regular” urls.

For instance, here's a smiley:

```

```

The browser will decode the string and show the image: 😊

To transform a blob into base64, we'll use the built-in `FileReader` object. It can read data from Blobs in multiple formats. In the [next chapter](#) we'll cover it more in-depth.

Here's the demo of downloading a blob, now via base-64:

```
let link = document.createElement('a');
link.download = 'hello.txt';
```

```
let blob = new Blob(['Hello, world!'], {type: 'text/plain'});

let reader = new FileReader();
reader.readAsDataURL(blob); // converts the blob to base64 and calls onload

reader.onload = function() {
  link.href = reader.result; // data url
  link.click();
};
```

Both ways of making an URL of a blob are usable. But usually `URL.createObjectURL(blob)` is simpler and faster.

URL.createObjectURL(blob)

- We need to revoke them if care about memory.
- Direct access to blob, no “encoding/decoding”

Blob to data url

- No need to revoke anything.
- Performance and memory losses on big blobs for encoding.

Image to blob

We can create a blob of an image, an image part, or even make a page screenshot. That’s handy to upload it somewhere.

Image operations are done via `<canvas>` element:

1. Draw an image (or its part) on canvas using [`canvas.drawImage`](#) .
2. Call canvas method [`.toBlob\(callback, format, quality\)`](#) that creates a blob and runs `callback` with it when done.

In the example below, an image is just copied, but we could cut from it, or transform it on canvas prior to making a blob:

```
// take any image
let img = document.querySelector('img');

// make <canvas> of the same size
let canvas = document.createElement('canvas');
canvas.width = img.clientWidth;
canvas.height = img.clientHeight;

let context = canvas.getContext('2d');
```

```
// copy image to it (this method allows to cut image)
context.drawImage(img, 0, 0);
// we can context.rotate(), and do many other things on canvas

// toBlob is async operation, callback is called when done
canvas.toBlob(function(blob) {
  // blob ready, download it
  let link = document.createElement('a');
  link.download = 'example.png';

  link.href = URL.createObjectURL(blob);
  link.click();

  // delete the internal blob reference, to let the browser clear memory from it
  URL.revokeObjectURL(link.href);
}, 'image/png');
```

If we prefer `async/await` instead of callbacks:

```
let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));
```

For screenshotting a page, we can use a library such as

<https://github.com/niklasvh/html2canvas> . What it does is just walks the page and draws it on `<canvas>` . Then we can get a blob of it the same way as above.

From Blob to ArrayBuffer

The `Blob` constructor allows to create a blob from almost anything, including any `BufferSource` .

But if we need to perform low-level processing, we can get the lowest-level `ArrayBuffer` from it using `FileReader` :

```
// get arrayBuffer from blob
let fileReader = new FileReader();

fileReader.readAsArrayBuffer(blob);

fileReader.onload = function(event) {
  let arrayBuffer = fileReader.result;
};
```

Summary

While `ArrayBuffer`, `Uint8Array` and other `BufferSource` are “binary data”, a `Blob` [↗](#) represents “binary data with type”.

That makes Blobs convenient for upload/download operations, that are so common in the browser.

Methods that perform web-requests, such as `XMLHttpRequest`, `fetch` and so on, can work with `Blob` natively, as well as with other binary types.

We can easily convert between `Blob` and low-level binary data types:

- We can make a Blob from a typed array using `new Blob(...)` constructor.
- We can get back `ArrayBuffer` from a Blob using `FileReader`, and then create a view over it for low-level binary processing.

File and FileReader

A `File` [↗](#) object inherits from `Blob` and is extended with filesystem-related capabilities.

There are two ways to obtain it.

First, there's a constructor, similar to `Blob`:

```
new File(fileParts, fileName, [options])
```

- **fileParts** – is an array of `Blob/BufferSource/String` values.
- **fileName** – file name string.
- **options** – optional object:
 - **lastModified** – the timestamp (integer date) of last modification.

Second, more often we get a file from `<input type="file">` or drag'n'drop or other browser interfaces. In that case, the file gets this information from OS.

As `File` inherits from `Blob`, `File` objects have the same properties, plus:

- **name** – the file name,
- **lastModified** – the timestamp of last modification.

That's how we can get a `File` object from `<input type="file">`:

```
<input type="file" onchange="showFile(this)">

<script>
function showFile(input) {
```

```
let file = input.files[0];

alert(`File name: ${file.name}`); // e.g my.png
alert(`Last modified: ${file.lastModified}`); // e.g 1552830408824
}
</script>
```

i Please note:

The input may select multiple files, so `input.files` is an array-like object with them. Here we have only one file, so we just take `input.files[0]`.

FileReader

[FileReader](#) is an object with the sole purpose of reading data from `Blob` (and hence `File` too) objects.

It delivers the data using events, as reading from disk may take time.

The constructor:

```
let reader = new FileReader(); // no arguments
```

The main methods:

- **`readAsArrayBuffer(blob)`** – read the data in binary format `ArrayBuffer`.
- **`readAsText(blob, [encoding])`** – read the data as a text string with the given encoding (`utf-8` by default).
- **`readAsDataURL(blob)`** – read the binary data and encode it as base64 data url.
- **`abort()`** – cancel the operation.

The choice of `read*` method depends on which format we prefer, how we're going to use the data.

- `readAsArrayBuffer` – for binary files, to do low-level binary operations. For high-level operations, like slicing, `File` inherits from `Blob`, so we can call them directly, without reading.
- `readAsText` – for text files, when we'd like to get a string.
- `readAsDataURL` – when we'd like to use this data in `src` for `img` or another tag. There's an alternative to reading a file for that, as discussed in chapter [Blob](#): `URL.createObjectURL(file)`.

As the reading proceeds, there are events:

- `loadstart` – loading started.
- `progress` – occurs during reading.
- `load` – no errors, reading complete.
- `abort` – `abort()` called.
- `error` – error has occurred.
- `loadend` – reading finished with either success or failure.

When the reading is finished, we can access the result as:

- `reader.result` is the result (if successful)
- `reader.error` is the error (if failed).

The most widely used events are for sure `load` and `error`.

Here's an example of reading a file:

```
<input type="file" onchange="readFile(this)">

<script>
function readFile(input) {
  let file = input.files[0];

  let reader = new FileReader();

  reader.readAsText(file);

  reader.onload = function() {
    console.log(reader.result);
  };

  reader.onerror = function() {
    console.log(reader.error);
  };
}
</script>
```

i `FileReader` for blobs

As mentioned in the chapter [Blob](#), `FileReader` can read not just files, but any blobs.

We can use it to convert a blob to another format:

- `readAsArrayBuffer(blob)` – to `ArrayBuffer`,
- `readAsText(blob, [encoding])` – to string (an alternative to `TextDecoder`),
- `readAsDataURL(blob)` – to base64 data url.

i `FileReaderSync` is available inside Web Workers

For Web Workers, there also exists a synchronous variant of `FileReader`, called [FileReaderSync](#) [↗](#).

Its reading methods `read*` do not generate events, but rather return a result, as regular functions do.

That's only inside a Web Worker though, because delays in synchronous calls, that are possible while reading from files, in Web Workers are less important. They do not affect the page.

Summary

`File` objects inherit from `Blob`.

In addition to `Blob` methods and properties, `File` objects also have `name` and `lastModified` properties, plus the internal ability to read from filesystem. We usually get `File` objects from user input, like `<input>` or Drag'n'Drop events (`ondragend`).

`FileReader` objects can read from a file or a blob, in one of three formats:

- String (`readAsText`).
- `ArrayBuffer` (`readAsArrayBuffer`).
- Data url, base-64 encoded (`readAsDataURL`).

In many cases though, we don't have to read the file contents. Just as we did with blobs, we can create a short url with `URL.createObjectURL(file)` and assign it to `<a>` or ``. This way the file can be downloaded or shown up as an image, as a part of canvas etc.

And if we're going to send a `File` over a network, that's also easy: network API like `XMLHttpRequest` or `fetch` natively accepts `File` objects.

Network requests

Fetch

JavaScript can send network requests to the server and load new information whenever is needed.

For example, we can:

- Submit an order,
- Load user information,
- Receive latest updates from the server,
- ...etc.

...And all of that without reloading the page!

There's an umbrella term "AJAX" (abbreviated **A**synchronous **J**avascript **A**nd **X**ml) for that. We don't have to use XML though: the term comes from old times, that's that word is there.

There are multiple ways to send a network request and get information from the server.

The `fetch()` method is modern and versatile, so we'll start with it. It evolved for several years and continues to improve, right now the support is pretty solid among browsers.

The basic syntax is:

```
let promise = fetch(url, [options])
```

- `url` – the URL to access.
- `options` – optional parameters: method, headers etc.

The browser starts the request right away and returns a `promise`.

Getting a response is usually a two-stage process.

First, the `promise` resolves with an object of the built-in `Response` [class](#) as soon as the server responds with headers.

So we can check HTTP status, to see whether it is successful or not, check headers, but don't have the body yet.

The promise rejects if the `fetch` was unable to make HTTP-request, e.g. network problems, or there's no such site. HTTP-errors, even such as 404 or 500, are considered a normal flow.

We can see them in response properties:

- **ok** – boolean, `true` if the HTTP status code is 200-299.
- **status** – HTTP status code.

For example:

```
let response = await fetch(url);

if (response.ok) { // if HTTP-status is 200-299
  // get the response body (see below)
  let json = await response.json();
} else {
  alert("HTTP-Error: " + response.status);
}
```

Second, to get the response body, we need to use an additional method call.

`Response` provides multiple promise-based methods to access the body in various formats:

- **response.json()** – parse the response as JSON object,
- **response.text()** – return the response as text,
- **response.formData()** – return the response as `FormData` object (form/multipart encoding, explained in the [next chapter](#)),
- **response.blob()** – return the response as `Blob` (binary data with type),
- **response.arrayBuffer()** – return the response as `ArrayBuffer` (pure binary data),
- additionally, `response.body` is a [ReadableStream](#) object, it allows to read the body chunk-by-chunk, we'll see an example later.

For instance, let's get a JSON-object with latest commits from GitHub:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript-tutorial/commits');
let commits = await response.json(); // read response body and parse as JSON

alert(commits[0].author.login);
```

Or, the same without `await`, using pure promises syntax:

```
fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits')
  .then(response => response.json())
  .then(commits => alert(commits[0].author.login));
```

To get the text, `await response.text()` instead of `.json()`:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits');
let text = await response.text(); // read response body as text
alert(text.slice(0, 80) + '...');
```

As a show-case for reading in binary format, let's fetch and show an image (see chapter [Blob](#) for details about operations on blobs):

```
let response = await fetch('/article/fetch/logo-fetch.svg');
let blob = await response.blob(); // download as Blob object

// create <img> for it
let img = document.createElement('img');
img.style = 'position:fixed;top:10px;left:10px;width:100px';
document.body.append(img);

// show it
img.src = URL.createObjectURL(blob);

setTimeout(() => { // hide after three seconds
  img.remove();
  URL.revokeObjectURL(img.src);
}, 3000);
```

Important:

We can choose only one body-parsing method.

If we got the response with `response.text()`, then `response.json()` won't work, as the body content has already been processed.

```
let text = await response.text(); // response body consumed
let parsed = await response.json(); // fails (already consumed)
```

Headers

There's a Map-like headers object in `response.headers`.

We can get individual headers or iterate over them:

```
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript-tutorial/issue-tracker');

// get one header
alert(response.headers.get('Content-Type')); // application/json; charset=utf-8

// iterate over all headers
for (let [key, value] of response.headers) {
  alert(`${key} = ${value}`);
}
```

To set a header, we can use the `headers` option, like this:

```
let response = fetch(protectedUrl, {
  headers: {
    Authentication: 'abcdef'
  }
});
```

...But there's a list of [forbidden HTTP headers](#) that we can't set:

- Accept-Charset , Accept-Encoding
- Access-Control-Request-Headers
- Access-Control-Request-Method
- Connection
- Content-Length
- Cookie , Cookie2
- Date
- DNT
- Expect
- Host
- Keep-Alive
- Origin
- Referer
- TE
- Trailer

- Transfer-Encoding
- Upgrade
- Via
- Proxy-*
- Sec-*

These headers ensure proper and safe HTTP, so they are controlled exclusively by the browser.

POST requests

To make a POST request, or a request with another method, we need to use `fetch` options:

- **method** – HTTP-method, e.g. POST,
- **body** – one of:
 - a string (e.g. JSON),
 - `FormData` object, to submit the data as `form/multipart`,
 - `Blob/BufferSource` to send binary data,
 - [URLSearchParams](#), to submit the data in `x-www-form-urlencoded` encoding, rarely used.

For example, this code submits `user` object as JSON:

```
let user = {
  name: 'John',
  surname: 'Smith'
};

let response = await fetch('/article/fetch/post/user', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json;charset=utf-8'
  },
  body: JSON.stringify(user)
});

let result = await response.json();
alert(result.message);
```

Please note, if the body is a string, then `Content-Type` is set to `text/plain;charset=UTF-8` by default. So we use `headers` option to send

`application/json` instead, that's the correct content type for JSON-encoded data.

Sending an image

We can also submit binary data directly using `Blob` or `BufferSource`.

For example, here's a `<canvas>` where we can draw by moving a mouse. A click on the "submit" button sends the image to server:

```
<body style="margin:0">
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>

  <input type="button" value="Submit" onclick="submit()">

  <script>
    canvasElem.onmousemove = function(e) {
      let ctx = canvasElem.getContext('2d');
      ctx.lineTo(e.clientX, e.clientY);
      ctx.stroke();
    };

    async function submit() {
      let blob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'
      let response = await fetch('/article/fetch/post/image', {
        method: 'POST',
        body: blob
      });
      let result = await response.json();
      alert(result.message);
    }

  </script>
</body>
```



Submit

Here we also didn't need to set `Content-Type` manually, because a `Blob` object has a built-in type (here `image/png`, as generated by `toBlob`).

The `submit()` function can be rewritten without `async/await` like this:

```
function submit() {
  canvasElem.toBlob(function(blob) {
    fetch('/article/fetch/post/image', {
      method: 'POST',
```

```

        body: blob
      })
      .then(response => response.json())
      .then(result => alert(JSON.stringify(result, null, 2)))
    }, 'image/png');
  }
}

```

Summary

A typical fetch request consists of two `await` calls:

```

let response = await fetch(url, options); // resolves with response headers
let result = await response.json(); // read body as json

```

Or, promise-style:

```

fetch(url, options)
  .then(response => response.json())
  .then(result => /* process result */)

```

Response properties:

- `response.status` – HTTP code of the response,
- `response.ok` – `true` if the status is 200-299.
- `response.headers` – Map-like object with HTTP headers.

Methods to get response body:

- `response.json()` – parse the response as JSON object,
- `response.text()` – return the response as text,
- `response.formData()` – return the response as `FormData` object (form/multipart encoding, see the next chapter),
- `response.blob()` – return the response as `Blob` (binary data with type),
- `response.arrayBuffer()` – return the response as `ArrayBuffer` (pure binary data),

Fetch options so far:

- `method` – HTTP-method,
- `headers` – an object with request headers (not any header is allowed),
- `body` – `string`, `FormData`, `BufferSource`, `Blob` or `UrlSearchParams` object to send.

In the next chapters we'll see more options and use cases of `fetch`.

✓ Tasks

Fetch users from GitHub

Create an async function `getUsers(names)`, that gets an array of GitHub logins, fetches the users from GitHub and returns an array of GitHub users.

The GitHub url with user information for the given `USERNAME` is:

`https://api.github.com/users/USERNAME`.

There's a test example in the sandbox.

Important details:

1. There should be one `fetch` request per user. And requests shouldn't wait for each other. So that the data arrives as soon as possible.
2. If any request fails, or if there's no such user, the function should return `null` in the resulting array.

[Open a sandbox with tests.](#) ↗

[To solution](#)

FormData

This chapter is about sending HTML forms: with or without files, with additional fields and so on. `FormData` ↗ objects can help with that.

The constructor is:

```
let formData = new FormData([form]);
```

If HTML `form` element is provided, it automatically captures its fields. As you may have already guessed, `FormData` is an object to store and send form data.

The special thing about `FormData` is that network methods, such as `fetch`, can accept a `FormData` object as a body. It's encoded and sent out with `Content-Type: form/multipart`. So, from the server point of view, that looks like a usual form submission.

Sending a simple form

Let's send a simple form first.

As you can see, that's almost one-liner:

```
<form id="formElem">
  <input type="text" name="name" value="John">
  <input type="text" name="surname" value="Smith">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('/article/formdata/post/user', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    alert(result.message);
  };
</script>
```

John Smith Submit

Here, the server accepts the POST request with the form and replies “User saved”.

FormData Methods

We can modify fields in `FormData` with methods:

- `formData.append(name, value)` – add a form field with the given `name` and `value`,
- `formData.append(name, blob, fileName)` – add a field as if it were `<input type="file">`, the third argument `fileName` sets file name (not form field name), as if it were a name of the file in user's filesystem,
- `formData.delete(name)` – remove the field with the given `name`,
- `formData.get(name)` – get the value of the field with the given `name`,
- `formData.has(name)` – if there exists a field with the given `name`, returns `true`, otherwise `false`

A form is technically allowed to have many fields with the same `name`, so multiple calls to `append` add more same-named fields.

There's also method `set`, with the same syntax as `append`. The difference is that `.set` removes all fields with the given `name`, and then appends a new field. So it makes sure there's only field with such `name`:

- `formData.set(name, value),`
- `formData.set(name, blob, fileName).`

Also we can iterate over `formData` fields using `for...of` loop:

```
let formData = new FormData();
formData.append('key1', 'value1');
formData.append('key2', 'value2');

// List key/value pairs
for(let [name, value] of formData) {
  alert(`${name} = ${value}`); // key1=value1, then key2=value2
}
```

Sending a form with a file

The form is always sent as `Content-Type: form/multipart`, this encoding allows to send files. So, `<input type="file">` fields are sent also, similar to a usual form submission.

Here's an example with such form:

```
<form id="formElem">
  <input type="text" name="firstName" value="John">
  Picture: <input type="file" name="picture" accept="image/*">
  <input type="submit">
</form>

<script>
  formElem.onsubmit = async (e) => {
    e.preventDefault();

    let response = await fetch('/article/formdata/post/user-avatar', {
      method: 'POST',
      body: new FormData(formElem)
    });

    let result = await response.json();

    alert(result.message);
```

```
};  
</script>
```

Picture:

No file chosen

Sending a form with Blob data

As we've seen in the chapter [Fetch](#), sending a dynamically generated `Blob`, e.g. an image, is easy. We can supply it directly as `fetch` parameter `body`.

In practice though, it's often convenient to send an image not separately, but as a part of the form, with additional fields, such as "name" and other metadata.

Also, servers are usually more suited to accept multipart-encoded forms, rather than raw binary data.

This example submits an image from `<canvas>`, along with some other fields, using `FormData`:

```
<body style="margin:0">  
  <canvas id="canvasElem" width="100" height="80" style="border:1px solid"></canvas>  
  
  <input type="button" value="Submit" onclick="submit()">  
  
  <script>  
    canvasElem.onmousemove = function(e) {  
      let ctx = canvasElem.getContext('2d');  
      ctx.lineTo(e.clientX, e.clientY);  
      ctx.stroke();  
    };  
  
    async function submit() {  
      let imageBlob = await new Promise(resolve => canvasElem.toBlob(resolve, 'image/png'));  
  
      let formData = new FormData();  
      formData.append("firstName", "John");  
      formData.append("image", imageBlob, "image.png");  
  
      let response = await fetch('/article/formdata/post/image-form', {  
        method: 'POST',  
        body: formData  
      });  
      let result = await response.json();  
      alert(result.message);  
    }  
  
  </script>  
</body>
```

Submit

Please note how the image `Blob` is added:

```
formData.append("image", imageBlob, "image.png");
```

That's same as if there were `<input type="file" name="image">` in the form, and the visitor submitted a file named `image.png` (3rd argument) from their filesystem.

Summary

[FormData](#) [↗](#) objects are used to capture HTML form and submit it using `fetch` or another network method.

We can either create `new FormData(form)` from an HTML form, or create an empty object, and then append fields with methods:

- `formData.append(name, value)`
- `formData.append(name, blob, fileName)`
- `formData.set(name, value)`
- `formData.set(name, blob, fileName)`

Two peculiarities here:

1. The `set` method removes fields with the same name, `append` doesn't.
2. To send a file, 3-argument syntax is needed, the last argument is a file name, that normally is taken from user filesystem for `<input type="file">`.

Other methods are:

- `formData.delete(name)`
- `formData.get(name)`
- `formData.has(name)`

That's it!

Fetch: Download progress

The `fetch` method allows to track *download* progress.

Please note: there's currently no way for `fetch` to track *upload* progress. For that purpose, please use [XMLHttpRequest](#), we'll cover it later.

To track download progress, we can use `response.body` property. It's a "readable stream" – a special object that provides body chunk-by-chunk, as it comes.

Unlike `response.text()`, `response.json()` and other methods, `response.body` gives full control over the reading process, and we can count how much is consumed at any moment.

Here's the sketch of code that reads the response from `response.body`:

```
// instead of response.json() and other methods
const reader = response.body.getReader();

// infinite loop while the body is downloading
while(true) {
  // done is true for the last chunk
  // value is Uint8Array of the chunk bytes
  const {done, value} = await reader.read();

  if (done) {
    break;
  }

  console.log(`Received ${value.length} bytes`)
}
```

The result of `await reader.read()` call is an object with two properties:

- **done** – true when the reading is complete.
- **value** – a typed array of bytes: `Uint8Array`.

We wait for more chunks in the loop, until `done` is `true`.

To log the progress, we just need for every `value` add its length to the counter.

Here's the full code to get response and log the progress, more explanations follow:

```
// Step 1: start the fetch and obtain a reader
let response = await fetch('https://api.github.com/repos/javascript-tutorial/en.javascript-tutorial/branches/main/src/index.html');

const reader = response.body.getReader();

// Step 2: get total length
const contentLength = +response.headers.get('Content-Length');

// Step 3: read the data
let receivedLength = 0; // length at the moment
```

```

let chunks = []; // array of received binary chunks (comprises the body)
while(true) {
  const {done, value} = await reader.read();

  if (done) {
    break;
  }

  chunks.push(value);
  receivedLength += value.length;

  console.log(`Received ${receivedLength} of ${contentLength}`)
}

// Step 4: concatenate chunks into single Uint8Array
let chunksAll = new Uint8Array(receivedLength); // (4.1)
let position = 0;
for(let chunk of chunks) {
  chunksAll.set(chunk, position); // (4.2)
  position += chunk.length;
}

// Step 5: decode into a string
let result = new TextDecoder("utf-8").decode(chunksAll);

// We're done!
let commits = JSON.parse(result);
alert(commits[0].author.login);

```

Let's explain that step-by-step:

1. We perform `fetch` as usual, but instead of calling `response.json()`, we obtain a stream reader `response.body.getReader()`.

Please note, we can't use both these methods to read the same response. Either use a reader or a response method to get the result.

2. Prior to reading, we can figure out the full response length from the `Content-Length` header.

It may be absent for cross-domain requests (see chapter [Fetch: Cross-Origin Requests](#)) and, well, technically a server doesn't have to set it. But usually it's at place.

3. Call `await reader.read()` until it's done.

We gather response `chunks` in the array. That's important, because after the response is consumed, we won't be able to "re-read" it using `response.json()` or another way (you can try, there'll be an error).

4. At the end, we have `chunks` – an array of `Uint8Array` byte chunks. We need to join them into a single result. Unfortunately, there's no single method that

concatenates those, so there's some code to do that:

1. We create `new Uint8Array(receivedLength)` – a same-typed array with the combined length.
2. Then use `.set(chunk, position)` method to copy each `chunk` one after another in the resulting array.
5. We have the result in `chunksAll`. It's a byte array though, not a string.

To create a string, we need to interpret these bytes. The built-in `TextDecoder` does exactly that. Then we can `JSON.parse` it.

What if we need binary content instead of JSON? That's even simpler. Replace steps 4 and 5 with a single call to a blob from all chunks:

```
let blob = new Blob(chunks);
```

At the end we have the result (as a string or a blob, whatever is convenient), and progress-tracking in the process.

Once again, please note, that's not for *upload* progress (no way now with `fetch`), only for *download* progress.

Fetch: Abort

Aborting a `fetch` is a little bit tricky. Remember, `fetch` returns a promise. And JavaScript generally has no concept of “aborting” a promise. So how can we cancel a fetch?

There's a special built-in object for such purposes: `AbortController`.

The usage is pretty simple:

- Step 1: create a controller:

```
let controller = new AbortController();
```

A controller is an extremely simple object. It has a single method `abort()`, and a single property `signal`. When `abort()` is called, the `abort` event triggers on `controller.signal`:

Like this:

```
let controller = new AbortController();  
let signal = controller.signal;
```

```
// triggers when controller.abort() is called
signal.addEventListener('abort', () => alert("abort!"));

controller.abort(); // abort!

alert(signal.aborted); // true (after abort)
```

- Step 2: pass the `signal` property to `fetch` option:

```
let controller = new AbortController();
fetch(url, {
  signal: controller.signal
});
```

Now `fetch` listens to the signal.

- Step 3: to abort, call `controller.abort()`:

```
controller.abort();
```

We're done: `fetch` gets the event from `signal` and aborts the request.

When a fetch is aborted, its promise rejects with an error named `AbortError`, so we should handle it:

```
// abort in 1 second
let controller = new AbortController();
setTimeout(() => controller.abort(), 1000);

try {
  let response = await fetch('/article/fetch-abort/demo/hang', {
    signal: controller.signal
  });
} catch(err) {
  if (err.name === 'AbortError') { // handle abort()
    alert("Aborted!");
  } else {
    throw err;
  }
}
```

`AbortController` is scalable, it allows to cancel multiple fetches at once.

For instance, here we fetch many `urls` in parallel, and the controller aborts them all:

```
let urls = [...]; // a list of urls to fetch in parallel

let controller = new AbortController();

let fetchJobs = urls.map(url => fetch(url, {
  signal: controller.signal
}));

let results = await Promise.all(fetchJobs);

// from elsewhere:
// controller.abort() stops all fetches
```

If we have our own jobs, different from `fetch`, we can use a single `AbortController` to stop those, together with fetches.

```
let urls = [...];
let controller = new AbortController();

let ourJob = new Promise((resolve, reject) => {
  ...
  controller.signal.addEventListener('abort', reject);
});

let fetchJobs = urls.map(url => fetch(url, {
  signal: controller.signal
}));

let results = await Promise.all([...fetchJobs, ourJob]);

// from elsewhere:
// controller.abort() stops all fetches and ourJob
```

Fetch: Cross-Origin Requests

If we make a `fetch` from an arbitrary web-site, that will probably fail.

The core concept here is *origin* – a domain/port/protocol triplet.

Cross-origin requests – those sent to another domain (even a subdomain) or protocol or port – require special headers from the remote side. That policy is called “CORS”: Cross-Origin Resource Sharing.

For instance, let’s try fetching `http://example.com`:

```
try {
  await fetch('http://example.com');
```

```
} catch(err) {  
  alert(err); // Failed to fetch  
}
```

Fetch fails, as expected.

Why? A brief history

Because cross-origin restrictions protect the internet from evil hackers.

Seriously. Let's make a very brief historical digression.

For many years a script from one site could not access the content of another site.

That simple, yet powerful rule was a foundation of the internet security. E.g. a script from the page `hacker.com` could not access user's mailbox at `gmail.com`. People felt safe.

JavaScript also did not have any special methods to perform network requests at that time. It was a toy language to decorate a web page.

But web developers demanded more power. A variety of tricks were invented to work around the limitation.

Using forms

One way to communicate with another server was to submit a `<form>` there. People submitted it into `<iframe>`, just to stay on the current page, like this:

```
<!-- form target -->  
<iframe name="iframe"></iframe>  
  
<!-- a form could be dynamically generated and submitted by JavaScript -->  
<form target="iframe" method="POST" action="http://another.com/...">  
  ...  
</form>
```

So, it was possible to make a GET/POST request to another site, even without networking methods. But as it's forbidden to access the content of an `<iframe>` from another site, it wasn't possible to read the response.

As we can see, forms allowed to send data anywhere, but not receive the response. To be precise, there were actually tricks for that (required special scripts at both the `iframe` and the page), but let these dinosaurs rest in peace.

Using scripts

Another trick was to use a `<script src="http://another.com/...">` tag. A script could have any `src`, from any domain. But again – it was impossible to

access the raw content of such script.

If `another.com` intended to expose data for this kind of access, then a so-called “JSONP (JSON with padding)” protocol was used.

Let’s say we need to get the data from `http://another.com` this way:

1. First, in advance, we declare a global function to accept the data, e.g. `gotWeather`.

```
// 1. Declare the function to process the data
function gotWeather({ temperature, humidity }) {
  alert(`temperature: ${temperature}, humidity: ${humidity}`);
}
```

2. Then we make a `<script>` tag with `src="http://another.com/weather.json?callback=gotWeather"`, please note that the name of our function is its `callback` parameter.

```
let script = document.createElement('script');
script.src = `http://another.com/weather.json?callback=gotWeather`;
document.body.append(script);
```

3. The remote server dynamically generates a script that calls `gotWeather(...)` with the data it wants us to receive.

```
// The expected answer from the server looks like this:
gotWeather({
  temperature: 25,
  humidity: 78
});
```

4. When the remote script loads and executes, `gotWeather` runs, and, as it’s our function, we have the data.

That works, and doesn’t violate security, because both sides agreed to pass the data this way. And, when both sides agree, it’s definitely not a hack. There are still services that provide such access, as it works even for very old browsers.

After a while, networking methods appeared, such as `XMLHttpRequest`.

At first, cross-origin requests were forbidden. But as a result of long discussions, cross-domain requests were allowed, in a way that does not add any capabilities unless explicitly allowed by the server.

Simple requests

There are two types of cross-domain requests:

1. Simple requests.
2. All the others.

Simple Requests are, well, simpler to make, so let's start with them.

A [simple request](#) is a request that satisfies two conditions:

1. [Simple method](#) : GET, POST or HEAD
2. [Simple headers](#) – the only allowed custom headers are:
 - Accept ,
 - Accept - Language ,
 - Content - Language ,
 - Content - Type with the value application/x-www-form-urlencoded , multipart/form-data or text/plain .

Any other request is considered “non-simple”. For instance, a request with PUT method or with an API - Key HTTP-header does not fit the limitations.

The essential difference is that a “simple request” can be made with a `<form>` or a `<script>` , without any special methods.

So, even a very old server should be ready to accept a simple request.

Contrary to that, requests with non-standard headers or e.g. method DELETE can't be created this way. For a long time JavaScript was unable to do such requests. So an old server may assume that such requests come from a privileged source, “because a webpage is unable to send them”.

When we try to make a non-simple request, the browser sends a special “preflight” request that asks the server – does it agree to accept such cross-origin requests, or not?

And, unless the server explicitly confirms that with headers, a non-simple request is not sent.

Now we'll go into details. All of them serve a single purpose – to ensure that new cross-origin capabilities are only accessible with an explicit permission from the server.

CORS for simple requests

If a request is cross-origin, the browser always adds Origin header to it.

For instance, if we request `https://anywhere.com/request` from `https://javascript.info/page`, the headers will be like:

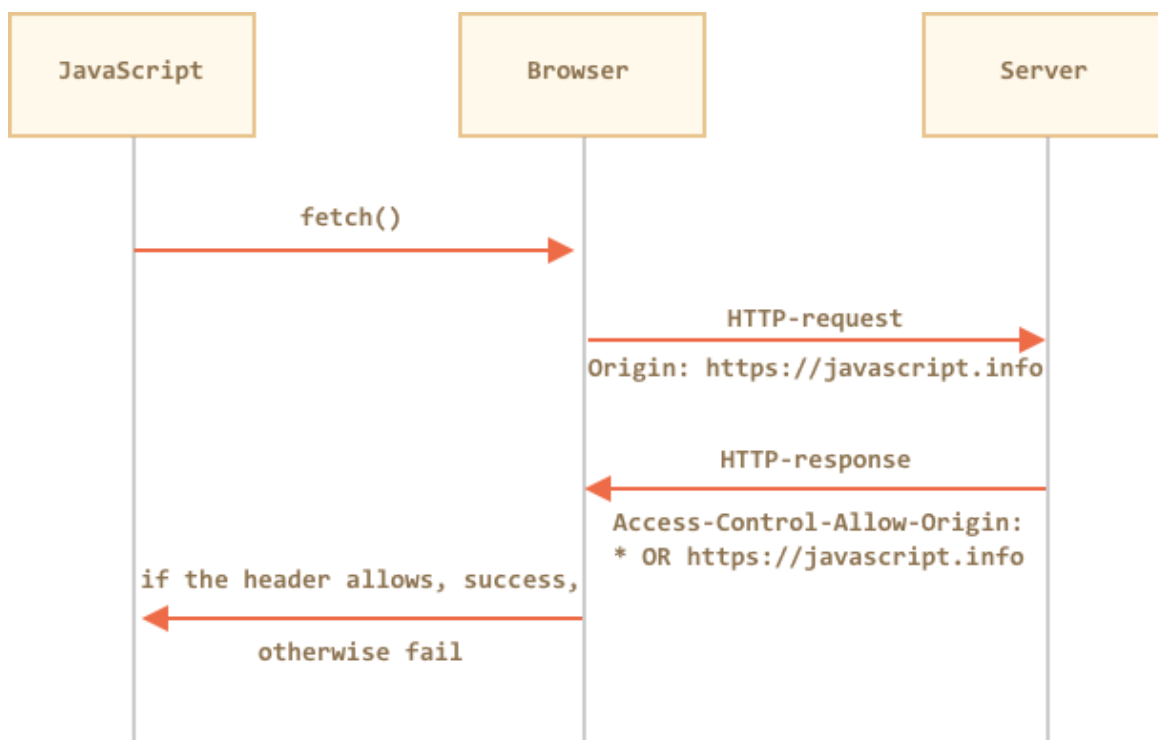
```
GET /request
Host: anywhere.com
Origin: https://javascript.info
...
```

As you can see, `Origin` contains exactly the origin (domain/protocol/port), without a path.

The server can inspect the `Origin` and, if it agrees to accept such a request, adds a special header `Access-Control-Allow-Origin` to the response. That header should contain the allowed origin (in our case `https://javascript.info`), or a star `*`. Then the response is successful, otherwise an error.

The browser plays the role of a trusted mediator here:

1. It ensures that the current `Origin` is sent with a cross-domain request.
2. It checks for correct `Access-Control-Allow-Origin` in the response, if it is so, then JavaScript access, otherwise forbids with an error.



Here's an example of a permissive server response:

```
200 OK
Content-Type:text/html; charset=UTF-8
```

Access-Control-Allow-Origin: https://javascript.info

Response headers

For cross-origin request, by default JavaScript may only access “simple response headers”:

- Cache-Control
- Content-Language
- Content-Type
- Expires
- Last-Modified
- Pragma

Any other response header is forbidden.

Please note: no Content-Length

Please note: there's no Content-Length header in the list!

This header contains the full response length. So, if we're downloading something and would like to track the percentage of progress, then an additional permission is required to access that header (see below).

To grant JavaScript access to any other response header, the server must list it in the Access-Control-Expose-Headers header.

For example:

```
200 OK
Content-Type:text/html; charset=UTF-8
Content-Length: 12345
API-Key: 2c9de507f2c54aa1
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Expose-Headers: Content-Length,API-Key
```

With such Access-Control-Expose-Headers header, the script is allowed to access Content-Length and API-Key headers of the response.

“Non-simple” requests

We can use any HTTP-method: not just GET/POST, but also PATCH, DELETE and others.

Some time ago no one could even assume that a webpage is able to do such requests. So there may exist webservices that treat a non-standard method as a signal: "That's not a browser". They can take it into account when checking access rights.

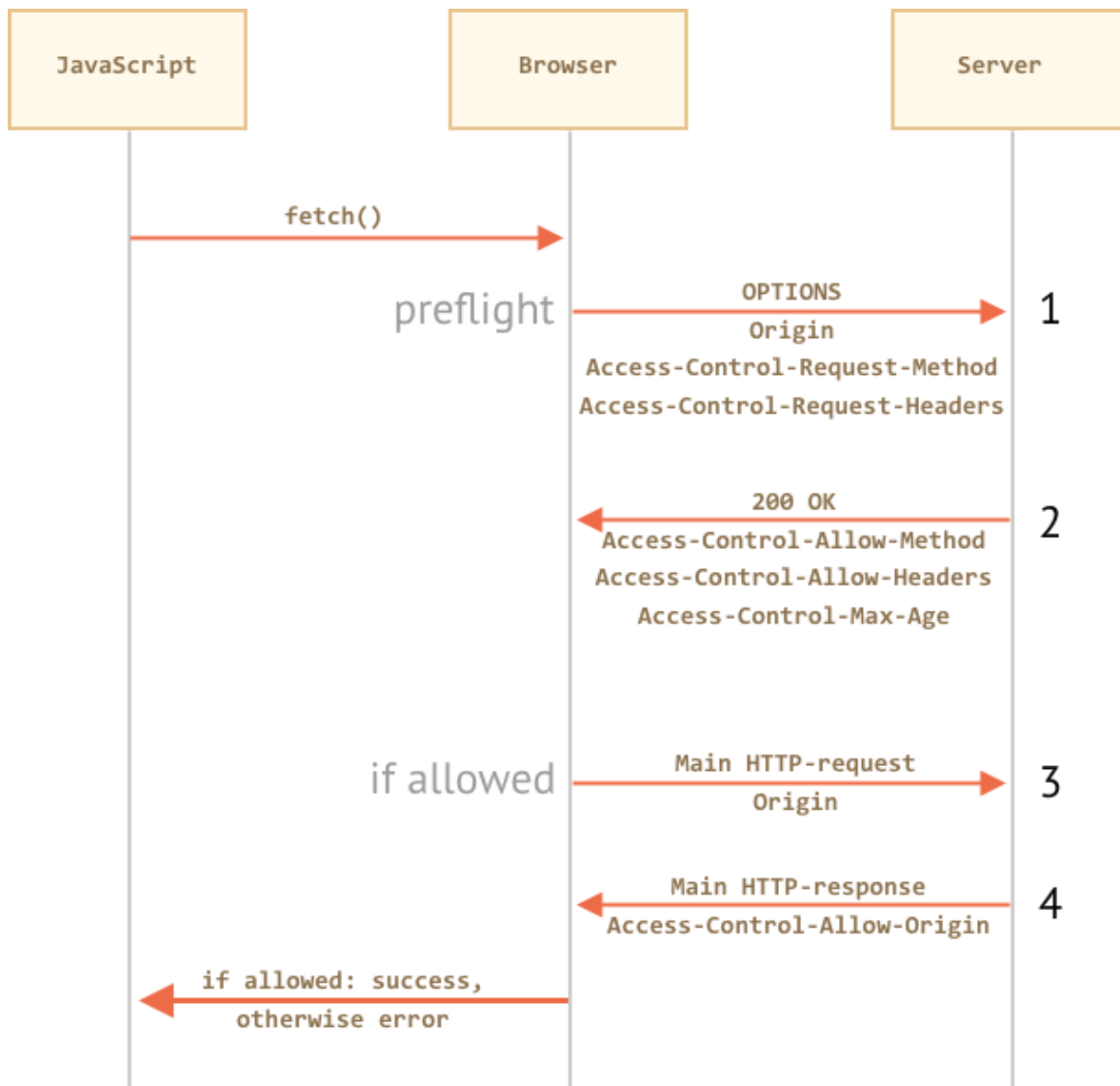
So, to avoid misunderstandings, any "non-simple" request – that couldn't be done in the old times, the browser does not make such requests right away. Before it sends a preliminary, so-called "preflight" request, asking for permission.

A preflight request uses method `OPTIONS` and has no body.

- `Access-Control-Request-Method` header has the requested method.
- `Access-Control-Request-Headers` header provides a comma-separated list of non-simple HTTP-headers.

If the server agrees to serve the requests, then it should respond with status 200, without body.

- The response header `Access-Control-Allow-Methods` must have the allowed method.
- The response header `Access-Control-Allow-Headers` must have a list of allowed headers.
- Additionally, the header `Access-Control-Max-Age` may specify a number of seconds to cache the permissions. So the browser won't have to send a preflight for subsequent requests that satisfy given permissions.



Let's see how it works step-by-step on example, for a cross-domain `PATCH` request (this method is often used to update data):

```
let response = await fetch('https://site.com/service.json', {
  method: 'PATCH',
  headers: {
    'Content-Type': 'application/json'
    'API-Key': 'secret'
  }
});
```

There are three reasons why the request is not simple (one is enough):

- Method `PATCH`
- `Content-Type` is not one of: `application/x-www-form-urlencoded`, `multipart/form-data`, `text/plain`.
- "Non-simple" `API-Key` header.

Step 1 (preflight request)

Prior to sending our request, the browser, on its own, sends a preflight request that looks like this:

```
OPTIONS /service.json
Host: site.com
Origin: https://javascript.info
Access-Control-Request-Method: PATCH
Access-Control-Request-Headers: Content-Type,API-Key
```

- Method: `OPTIONS`.
- The path – exactly the same as the main request: `/service.json`.
- Cross-origin special headers:
 - `Origin` – the source origin.
 - `Access-Control-Request-Method` – requested method.
 - `Access-Control-Request-Headers` – a comma-separated list of “non-simple” headers.

Step 2 (preflight response)

The server should respond with status 200 and headers:

- `Access-Control-Allow-Methods: PATCH`
- `Access-Control-Allow-Headers: Content-Type,API-Key`.

That allows future communication, otherwise an error is triggered.

If the server expects other methods and headers in the future, makes sense to allow them in advance by adding to the list:

```
200 OK
Access-Control-Allow-Methods: PUT,PATCH,DELETE
Access-Control-Allow-Headers: API-Key,Content-Type,If-Modified-Since,Cache-Control
Access-Control-Max-Age: 86400
```

Now the browser can see that `PATCH` is in the list of allowed methods, and both headers are in the list too, so it sends out the main request.

Besides, the preflight response is cached for time, specified by `Access-Control-Max-Age` header (86400 seconds, one day), so subsequent requests will not cause a preflight. Assuming that they fit the allowances, they will be sent directly.

Step 3 (actual request)

When the preflight is successful, the browser now makes the real request. Here the flow is the same as for simple requests.

The real request has `Origin` header (because it's cross-origin):

```
PATCH /service.json
Host: site.com
Content-Type: application/json
API-Key: secret
Origin: https://javascript.info
```

Step 4 (actual response)

The server should not forget to add `Access-Control-Allow-Origin` to the response. A successful preflight does not relieve from that:

```
Access-Control-Allow-Origin: https://javascript.info
```

Now everything's correct. JavaScript is able to read the full response.

Credentials

A cross-origin request by default does not bring any credentials (cookies or HTTP authentication).

That's uncommon for HTTP-requests. Usually, a request to `http://site.com` is accompanied by all cookies from that domain. But cross-domain requests made by JavaScript methods are an exception.

For example, `fetch('http://another.com')` does not send any cookies, even those that belong to `another.com` domain.

Why?

That's because a request with credentials is much more powerful than an anonymous one. If allowed, it grants JavaScript the full power to act and access sensitive information on behalf of a user.

Does the server really trust pages from `Origin` that much? Then it must explicitly allow requests with credentials with an additional header.

To send credentials, we need to add the option `credentials: "include"`, like this:

```
fetch('http://another.com', {
  credentials: "include"
});
```

Now `fetch` sends cookies originating from `another.com` without request to that site.

If the server wishes to accept the request with credentials, it should add a header `Access-Control-Allow-Credentials: true` to the response, in addition to `Access-Control-Allow-Origin`.

For example:

```
200 OK
Access-Control-Allow-Origin: https://javascript.info
Access-Control-Allow-Credentials: true
```

Please note: `Access-Control-Allow-Origin` is prohibited from using a star `*` for requests with credentials. There must be exactly the origin there, like above. That's an additional safety measure, to ensure that the server really knows who it trusts.

Summary

Networking methods split cross-origin requests into two kinds: “simple” and all the others.

[Simple requests](#)  must satisfy the following conditions:

- Method: GET, POST or HEAD.
- Headers – we can set only:
 - `Accept`
 - `Accept-Language`
 - `Content-Language`
 - `Content-Type` to the value `application/x-www-form-urlencoded`, `multipart/form-data` or `text/plain`.

The essential difference is that simple requests were doable since ancient times using `<form>` or `<script>` tags, while non-simple were impossible for browsers for a long time.

So, practical difference is that simple requests are sent right away, with `Origin` header, but for other ones the browser makes a preliminary “preflight” request, asking for permission.

For simple requests:

- → The browser sends `Origin` header with the origin.
- ← For requests without credentials (not sent default), the server should set:

- `Access-Control-Allow-Origin` to `*` or same value as `Origin`
- ← For requests with credentials, the server should set:
 - `Access-Control-Allow-Origin` to same value as `Origin`
 - `Access-Control-Allow-Credentials` to `true`

Additionally, if JavaScript wants to access non-simple response headers:

- `Cache-Control`
- `Content-Language`
- `Content-Type`
- `Expires`
- `Last-Modified`
- `Pragma`

...Then the server should list the allowed ones in `Access-Control-Expose-Headers` header.

For non-simple requests, a preliminary “preflight” request is issued before the requested one:

- → The browser sends `OPTIONS` request to the same url, with headers:
 - `Access-Control-Request-Method` has requested method.
 - `Access-Control-Request-Headers` lists non-simple requested headers
- ← The server should respond with status 200 and headers:
 - `Access-Control-Allow-Methods` with a list of allowed methods,
 - `Access-Control-Allow-Headers` with a list of allowed headers,
 - `Access-Control-Max-Age` with a number of seconds to cache permissions.
- Then the actual request is sent, the previous “simple” scheme is applied.

✓ Tasks

Why do we need Origin?

importance: 5

As you probably know, there’s HTTP-header `Referer` , that usually contains an url of the page which initiated a network request.

For instance, when fetching `http://google.com` from `http://javascript.info/some/url` , the headers look like this:


```
Accept: */*
Accept-Charset: utf-8
Accept-Encoding: gzip,deflate,sdch
Connection: keep-alive
Host: google.com
Origin: http://javascript.info
Referer: http://javascript.info/some/url
```

As you can see, both `Referer` and `Origin` are present.

The questions:

1. Why `Origin` is needed, if `Referer` has even more information?
2. If it possible that there's no `Referer` or `Origin`, or it's incorrect?

[To solution](#)

Fetch API

So far, we know quite a bit about `fetch`.

Now let's see the rest of API, to cover all its abilities.

Here's the full list of all possible `fetch` options with their default values (alternatives in comments):

```
let promise = fetch(url, {
  method: "GET", // POST, PUT, DELETE, etc.
  headers: {
    // the content type header value is usually auto-set depending on the request body
    "Content-Type": "text/plain;charset=UTF-8"
  },
  body: undefined // string, FormData, Blob, BufferSource, or URLSearchParams
  referrer: "about:client", // or "" to send no Referer header, or an url from the document
  referrerPolicy: "no-referrer-when-downgrade", // no-referrer, origin, same-origin
  mode: "cors", // same-origin, no-cors
  credentials: "same-origin", // omit, include
  cache: "default", // no-store, reload, no-cache, force-cache, or only-if-cached
  redirect: "follow", // manual, error
  integrity: "", // a hash, like "sha256-abcdef1234567890"
  keepalive: false, // true
  signal: undefined, // AbortController to abort request
  window: window // null
});
```

An impressive list, right?

We fully covered `method` , `headers` and `body` in the chapter [Fetch](#).

The `signal` option is covered in [Fetch: Abort](#).

Now let's explore the rest of options.

referrer, referrerPolicy

These options govern how `fetch` sets HTTP `Referer` header.

That header contains the url of the page that made the request. In most scenarios, it plays a very minor informational role, but sometimes, for security purposes, it makes sense to remove or shorten it.

The `referrer` option allows to set any `Referer` within the current origin) or disable it.

To send no referer, set an empty string:

```
fetch('/page', {  
  referrer: "" // no Referer header  
});
```

To set another url within the current origin:

```
fetch('/page', {  
  // assuming we're on https://javascript.info  
  // we can set any Referer header, but only within the current origin  
  referrer: "https://javascript.info/anotherpage"  
});
```

The `referrerPolicy` option sets general rules for `Referer` .

Possible values are described in the [Referrer Policy specification](#) ↗ :

- **"no-referrer-when-downgrade"** – default value: `Referer` is sent always, unless we send a request from HTTPS to HTTP (to less secure protocol).
- **"no-referrer"** – never send `Referer` .
- **"origin"** – only send the origin in `Referer` , not the full page URL, e.g. `http://site.com` instead of `http://site.com/path` .
- **"origin-when-cross-origin"** – send full `Referer` to the same origin, but only the origin part for cross-origin requests.
- **"same-origin"** – send full `Referer` to the same origin, but no referer for cross-origin requests.

- **"strict-origin"** – send only origin, don't send `Referer` for HTTPS → HTTP requests.
- **"strict-origin-when-cross-origin"** – for same-origin send full `Referer`, for cross-origin send only origin, unless it's HTTPS → HTTP request, then send nothing.
- **"unsafe-url"** – always send full url in `Referer`.

Let's say we have an admin zone with URL structure that shouldn't be known from outside of the site.

If we send a cross-origin `fetch`, then by default it sends the `Referer` header with the full url of our page (except when we request from HTTPS to HTTP, then no `Referer`).

E.g. `Referer: https://javascript.info/admin/secret/paths`.

If we'd like to totally hide the referrer:

```
fetch('https://another.com/page', {
  referrerPolicy: "no-referrer" // no Referer, same effect as referrer: ""
});
```

Otherwise, if we'd like the remote side to see only the domain where the request comes from, but not the full URL, we can send only the "origin" part of it:

```
fetch('https://another.com/page', {
  referrerPolicy: "strict-origin" // Referer: https://javascript.info
});
```

mode

The `mode` option serves as a safe-guard that prevents cross-origin requests:

- **"cors"** – the default, cross-origin requests are allowed, as described in [Fetch: Cross-Origin Requests](#),
- **"same-origin"** – cross-origin requests are forbidden,
- **"no-cors"** – only simple cross-origin requests are allowed.

That may be useful in contexts when the fetch url comes from 3rd-party, and we want a "power off switch" to limit cross-origin capabilities.

credentials

The `credentials` option specifies whether `fetch` should send cookies and HTTP-Authorization headers with the request.

- **"same-origin"** – the default, don't send for cross-origin requests,
- **"include"** – always send, requires `Accept-Control-Allow-Credentials` from cross-origin server,
- **"omit"** – never send, even for same-origin requests.

cache

By default, `fetch` requests make use of standard HTTP-caching. That is, it honors `Expires`, `Cache-Control` headers, sends `If-Modified-Since`, and so on. Just like regular HTTP-requests do.

The `cache` options allows to ignore HTTP-cache or fine-tune its usage:

- **"default"** – `fetch` uses standard HTTP-cache rules and headers;
- **"no-store"** – totally ignore HTTP-cache, this mode becomes the default if we set a header `If-Modified-Since`, `If-None-Match`, `If-Unmodified-Since`, `If-Match`, or `If-Range`;
- **"reload"** – don't take the result from HTTP-cache (if any), but populate cache with the response (if response headers allow);
- **"no-cache"** – create a conditional request if there is a cached response, and a normal request otherwise. Populate HTTP-cache with the response;
- **"force-cache"** – use a response from HTTP-cache, even if it's stale. If there's no response in HTTP-cache, make a regular HTTP-request, behave normally;
- **"only-if-cached"** – use a response from HTTP-cache, even if it's stale. If there's no response in HTTP-cache, then error. Only works when `mode` is `"same-origin"`.

redirect

Normally, `fetch` transparently follows HTTP-redirects, like 301, 302 etc.

The `redirect` option allows to change that:

- **"follow"** – the default, follow HTTP-redirects,
- **"error"** – error in case of HTTP-redirect,
- **"manual"** – don't follow HTTP-redirect, but `response.url` will be the new URL, and `response.redirected` will be `true`, so that we can perform the redirect manually to the new URL (if needed).

integrity

The `integrity` option allows to check if the response matches the known-ahead checksum.

As described in the [specification](#) [↗](#), supported hash-functions are SHA-256, SHA-384, and SHA-512, there might be others depending on a browser.

For example, we're downloading a file, and we know that it's SHA-256 checksum is "abc" (a real checksum is longer, of course).

We can put it in the `integrity` option, like this:

```
fetch('http://site.com/file', {  
  integrity: 'sha256-abd'  
});
```

Then `fetch` will calculate SHA-256 on its own and compare it with our string. In case of a mismatch, an error is triggered.

keepalive

The `keepalive` option indicates that the request may outlive the page.

For example, we gather statistics about how the current visitor uses our page (mouse clicks, page fragments he views), to improve user experience.

When the visitor leaves our page – we'd like to save it on our server.

We can use `window.onunload` for that:

```
window.onunload = function() {  
  fetch('/analytics', {  
    method: 'POST',  
    body: "statistics",  
    keepalive: true  
  });  
};
```

Normally, when a document is unloaded, all associated network requests are aborted. But `keepalive` option tells the browser to perform the request in background, even after it leaves the page. So it's essential for our request to succeed.

- We can't send megabytes: the body limit for keepalive requests is 64kb.
 - If we gather more data, we can send it out regularly, then there won't be a lot for the "onunload" request.

- The limit is for all currently ongoing requests. So we cheat it by creating 100 requests, each 64kb.
- We don't get the server response if the request is made `onunload`, because the document is already unloaded at that time.
 - Usually, the server sends empty response to such requests, so it's not a problem.

URL objects

The built-in [URL](#) class provides a convenient interface for creating and parsing URLs.

There are no networking methods that require exactly an `URL` object, strings are good enough. So technically we don't have to use `URL`. But sometimes it can be really helpful.

Creating an URL

The syntax to create a new URL object:

```
new URL(url, [base])
```

- **url** – the URL string or path (if base is set, see below).
- **base** – an optional base, if set and `url` has only path, then the URL is generated relative to `base`.

For example, these two URLs are same:

```
let url1 = new URL('https://javascript.info/profile/admin');
let url2 = new URL('/profile/admin', 'https://javascript.info');

alert(url1); // https://javascript.info/profile/admin
alert(url2); // https://javascript.info/profile/admin
```

Go to the path relative to the current URL:

```
let url = new URL('https://javascript.info/profile/admin');
let testerUrl = new URL('tester', url);

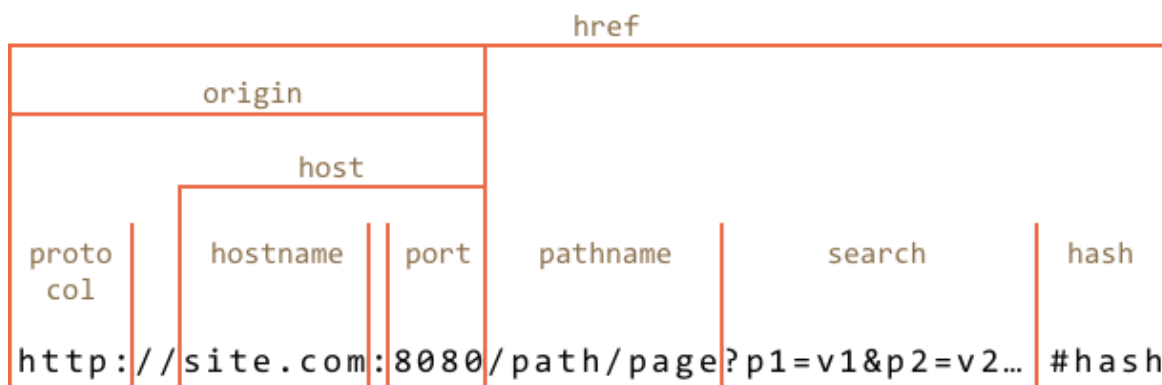
alert(testerUrl); // https://javascript.info/profile/tester
```

The `URL` object immediately allows us to access its components, so it's a nice way to parse the url, e.g.:

```
let url = new URL('https://javascript.info/url');

alert(url.protocol); // https:
alert(url.host);     // javascript.info
alert(url.pathname); // /url
```

Here's the cheatsheet:



- `href` is the full url, same as `url.toString()`
- `protocol` ends with the colon character `:`
- `search` – a string of parameters, starts with the question mark `?`
- `hash` starts with the hash character `#`
- there may be also `user` and `password` properties if HTTP authentication is present: `http://login:password@site.com` (not painted above, rarely used).

i We can use `URL` everywhere instead of a string

We can use an `URL` object in `fetch` or `XMLHttpRequest`, almost everywhere where a string url is expected.

In the vast majority of methods it's automatically converted to a string.

SearchParams “?...”

Let's say we want to create an url with given search params, for instance, `https://google.com/search?query=JavaScript`.

We can provide them in the URL string:

```
new URL('https://google.com/search?query=JavaScript')
```

...But parameters need to be encoded if they contain spaces, non-latin letters, etc (more about that below).

So there's URL property for that: `url.searchParams`, an object of type [URLSearchParams](#).

It provides convenient methods for search parameters:

- `append(name, value)` – add the parameter,
- `delete(name)` – remove the parameter,
- `get(name)` – get the parameter,
- `getAll(name)` – get all parameters with the same `name` (that's possible, e.g. `?user=John&user=Pete`),
- `has(name)` – check for the existence of the parameter,
- `set(name, value)` – set/replace the parameter,
- `sort()` – sort parameters by name, rarely needed,
- ...and also iterable, similar to `Map`.

For example:

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'test me!'); // added parameter with a space and !

alert(url); // https://google.com/search?query=test+me%21

url.searchParams.set('tbs', 'qdr:y'); // this parameter specifies for date range for

alert(url); // https://google.com/search?q=test+me%21&tbs=qdr:y

// iterate over search parameters (decoded)
for(let [name, value] of url.searchParams) {
  alert(`${name}=${value}`); // q=test me!, then tbs=qdr:y
}
```

Encoding

There's a standard [RFC3986](#) that defines which characters are allowed and which are not.

Those that are not allowed, must be encoded, for instance non-latin letters and spaces – replaced with their UTF-8 codes, prefixed by `%`, such as `%20` (a space can be encoded by `+`, for historical reasons that's allowed in URL too).

The good news is that `URL` objects handle all that automatically. We just supply all parameters unencoded, and then convert the URL to the string:

```
// using some cyrillic characters for this example

let url = new URL('https://ru.wikipedia.org/wiki/Тест');

url.searchParams.set('key', 'т');
alert(url); //https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82?key=%D1%8A
```

As you can see, both `Тест` in the url path and `т` in the parameter are encoded.

Encoding strings

If we're using strings instead of URL objects, then we can encode manually using built-in functions:

- [encodeURIComponent](#) – encode URL as a whole.
- [decodeURIComponent](#) – decode it back.
- [encodeURIComponent](#) – encode URL components, such as search parameters, or a hash, or a pathname.
- [decodeURIComponent](#) – decodes it back.

What's the difference between `encodeURIComponent` and `encodeURIComponent`?

That's easy to understand if we look at the URL, that's split into components in the picture above:

```
http://site.com:8080/path/page?p1=v1&p2=v2#hash
```

As we can see, characters such as `:`, `?`, `=`, `&`, `#` are allowed in URL. Some others, including non-latin letters and spaces, must be encoded.

That's what `encodeURIComponent` does:

```
// using cyrcillic characters in url path
let url = encodeURIComponent('http://site.com/привет');

// each cyrillic character is encoded with two %xx
// together they form UTF-8 code for the character
alert(url); // http://site.com/%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D1%82
```

...On the other hand, if we look at a single URL component, such as a search parameter, we should encode more characters, e.g. `?`, `=` and `&` are used for

formatting.

That's what `encodeURIComponent` does. It encodes same characters as `encodeURIComponent`, plus a lot of others, to make the resulting value safe to use in any URL component.

For example:

```
let music = encodeURIComponent('Rock&Roll');

let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock%26Roll
```

Compare with `encodeURIComponent`:

```
let music = encodeURIComponent('Rock&Roll');

let url = `https://google.com/search?q=${music}`;
alert(url); // https://google.com/search?q=Rock&Roll
```

As we can see, `encodeURIComponent` does not encode `&`, as this is a legit character in URL as a whole.

But we should encode `&` inside a search parameter, otherwise, we get `q=Rock&Roll` – that is actually `q=Rock` plus some obscure parameter `Roll`. Not as intended.

So we should use only `encodeURIComponent` for each search parameter, to correctly insert it in the URL string. The safest is to encode both name and value, unless we're absolutely sure that either has only allowed characters.

Why URL?

Lots of old code uses these functions, these are sometimes convenient, and by now means not dead.

But in modern code, it's recommended to use classes [URL](#) and [URLSearchParams](#).

One of the reason is: they are based on the recent URI spec: [RFC3986](#), while `encode*` functions are based on the obsolete version [RFC2396](#).

For example, IPv6 addresses are treated differently:

```
// valid url with IPv6 address
let url = 'http://[2607:f8b0:4005:802::1007]/';
```

```
alert(encodeURIComponent(url)); // http://%5B2607:f8b0:4005:802::1007%5D/  
alert(new URL(url)); // http://[2607:f8b0:4005:802::1007]/
```

As we can see, `encodeURIComponent` replaced square brackets `[. . .]`, that's not correct, the reason is: IPv6 urls did not exist at the time of RFC2396 (August 1998).

Such cases are rare, `encode*` functions work well most of the time, it's just one of the reason to prefer new APIs.

XMLHttpRequest

`XMLHttpRequest` is a built-in browser object that allows to make HTTP requests in JavaScript.

Despite of having the word “XML” in its name, it can operate on any data, not only in XML format. We can upload/download files, track progress and much more.

Right now, there's another, more modern method `fetch`, that somewhat deprecates `XMLHttpRequest`.

In modern web-development `XMLHttpRequest` may be used for three reasons:

1. Historical reasons: we need to support existing scripts with `XMLHttpRequest`.
2. We need to support old browsers, and don't want polyfills (e.g. to keep scripts tiny).
3. We need something that `fetch` can't do yet, e.g. to track upload progress.

Does that sound familiar? If yes, then all right, go on with `XMLHttpRequest`. Otherwise, please head on to [Fetch](#).

The basics

`XMLHttpRequest` has two modes of operation: synchronous and asynchronous.

Let's see the asynchronous first, as it's used in the majority of cases.

To do the request, we need 3 steps:

1. Create `XMLHttpRequest`:

```
let xhr = new XMLHttpRequest(); // the constructor has no arguments
```

2. Initialize it:

```
xhr.open(method, URL, [async, user, password])
```

This method is usually called right after `new XMLHttpRequest`. It specifies the main parameters of the request:

- `method` – HTTP-method. Usually `"GET"` or `"POST"`.
- `URL` – the URL to request, a string, can be [URL](#) object.
- `async` – if explicitly set to `false`, then the request is synchronous, we'll cover that a bit later.
- `user`, `password` – login and password for basic HTTP auth (if required).

Please note that `open` call, contrary to its name, does not open the connection. It only configures the request, but the network activity only starts with the call of `send`.

3. Send it out.

```
xhr.send([body])
```

This method opens the connection and sends the request to server. The optional `body` parameter contains the request body.

Some request methods like `GET` do not have a body. And some of them like `POST` use `body` to send the data to the server. We'll see examples later.

4. Listen to events for response.

These three are the most widely used:

- `load` – when the result is ready, that includes HTTP errors like 404.
- `error` – when the request couldn't be made, e.g. network down or invalid URL.
- `progress` – triggers periodically during the download, reports how much downloaded.

```
xhr.onload = function() {  
    alert(`Loaded: ${xhr.status} ${xhr.response}`);  
};  
  
xhr.onerror = function() { // only triggers if the request couldn't be made at all  
    alert(`Network Error`);  
};  
  
xhr.onprogress = function(event) { // triggers periodically  
    // event.loaded - how many bytes downloaded  
    // event.lengthComputable = true if the server sent Content-Length header  
    // event.total - total number of bytes (if lengthComputable)
```

```
    alert(`Received ${event.loaded} of ${event.total}`);  
  };
```

Here's a full example. The code below loads the URL at `/article/xmlhttprequest/example/load` from the server and prints the progress:

```
// 1. Create a new XMLHttpRequest object  
let xhr = new XMLHttpRequest();  
  
// 2. Configure it: GET-request for the URL /article/.../load  
xhr.open('GET', '/article/xmlhttprequest/example/load');  
  
// 3. Send the request over the network  
xhr.send();  
  
// 4. This will be called after the response is received  
xhr.onload = function() {  
  if (xhr.status !== 200) { // analyze HTTP status of the response  
    alert(`Error ${xhr.status}: ${xhr.statusText}`); // e.g. 404: Not Found  
  } else { // show the result  
    alert(`Done, got ${xhr.response.length} bytes`); // responseText is the server  
  }  
};  
  
xhr.onprogress = function(event) {  
  if (event.lengthComputable) {  
    alert(`Received ${event.loaded} of ${event.total} bytes`);  
  } else {  
    alert(`Received ${event.loaded} bytes`); // no Content-Length  
  }  
};  
  
xhr.onerror = function() {  
  alert("Request failed");  
};
```

Once the server has responded, we can receive the result in the following properties of the request object:

status

HTTP status code (a number): `200`, `404`, `403` and so on, can be `0` in case of a non-HTTP failure.

statusText

HTTP status message (a string): usually OK for 200, Not Found for 404, Forbidden for 403 and so on.

response (old scripts may use responseText)

The server response.

We can also specify a timeout using the corresponding property:

```
xhr.timeout = 10000; // timeout in ms, 10 seconds
```

If the request does not succeed within the given time, it gets canceled and `timeout` event triggers.

i URL search parameters

To pass URL parameters, like `?name=value`, and ensure the proper encoding, we can use `URL` object:

```
let url = new URL('https://google.com/search');
url.searchParams.set('q', 'test me!');

// the parameter 'q' is encoded
xhr.open('GET', url); // https://google.com/search?q=test+me%21
```

Response Type

We can use `xhr.responseType` property to set the response format:

- `"` (default) – get as string,
- `"text"` – get as string,
- `"arraybuffer"` – get as `ArrayBuffer` (for binary data, see chapter [ArrayBuffer, binary arrays](#)),
- `"blob"` – get as `Blob` (for binary data, see chapter [Blob](#)),
- `"document"` – get as XML document (can use XPath and other XML methods),
- `"json"` – get as JSON (parsed automatically).

For example, let's get the response as JSON:

```
let xhr = new XMLHttpRequest();
```

```
xhr.open('GET', '/article/xmlhttprequest/example/json');

xhr.responseType = 'json';

xhr.send();

// the response is {"message": "Hello, world!"}
xhr.onload = function() {
  let responseObj = xhr.response;
  alert(responseObj.message); // Hello, world!
};
```

Please note:

In the old scripts you may also find `xhr.responseText` and even `xhr.responseXML` properties.

They exist for historical reasons, to get either a string or XML document. Nowadays, we should set the format in `xhr.responseType` and get `xhr.response` as demonstrated above.

Ready states

`XMLHttpRequest` changes between states as it progresses. The current state is accessible as `xhr.readyState`.

All states, as in [the specification](#) :

```
UNSENT = 0; // initial state
OPENED = 1; // open called
HEADERS_RECEIVED = 2; // response headers received
LOADING = 3; // response is loading (a data packet is received)
DONE = 4; // request complete
```

An `XMLHttpRequest` object travels them in the order `0 → 1 → 2 → 3 → ... → 3 → 4`. State `3` repeats every time a data packet is received over the network.

We can track them using `readystatechange` event:

```
xhr.onreadystatechange = function() {
  if (xhr.readyState == 3) {
    // loading
  }
  if (xhr.readyState == 4) {
    // request finished
  }
};
```

```
}  
};
```

You can find `readystatechange` listeners in really old code, it's there for historical reasons, as there was a time when there were no `load` and other events. Nowadays, `load/error/progress` handlers deprecate it.

Aborting request

We can terminate the request at any time. The call to `xhr.abort()` does that:

```
xhr.abort(); // terminate the request
```

That triggers `abort` event, and `xhr.status` becomes `0`.

Synchronous requests

If in the `open` method the third parameter `async` is set to `false`, the request is made synchronously.

In other words, JavaScript execution pauses at `send()` and resumes when the response is received. Somewhat like `alert` or `prompt` commands.

Here's the rewritten example, the 3rd parameter of `open` is `false`:

```
let xhr = new XMLHttpRequest();  
  
xhr.open('GET', '/article/xmlhttprequest/hello.txt', false);  
  
try {  
  xhr.send();  
  if (xhr.status !== 200) {  
    alert(`Error ${xhr.status}: ${xhr.statusText}`);  
  } else {  
    alert(xhr.response);  
  }  
} catch(err) { // instead of onerror  
  alert("Request failed");  
}
```

It might look good, but synchronous calls are used rarely, because they block in-page JavaScript till the loading is complete. In some browsers it becomes impossible to scroll. If a synchronous call takes too much time, the browser may suggest to close the “hanging” webpage.

Many advanced capabilities of `XMLHttpRequest`, like requesting from another domain or specifying a timeout, are unavailable for synchronous requests. Also, as you can see, no progress indication.

Because of all that, synchronous requests are used very sparingly, almost never. We won't talk about them any more.

HTTP-headers

`XMLHttpRequest` allows both to send custom headers and read headers from the response.

There are 3 methods for HTTP-headers:

`setRequestHeader(name, value)`

Sets the request header with the given `name` and `value`.

For instance:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

Headers limitations

Several headers are managed exclusively by the browser, e.g. `Referer` and `Host`. The full list is [in the specification](#).

`XMLHttpRequest` is not allowed to change them, for the sake of user safety and correctness of the request.

Can't remove a header

Another peculiarity of `XMLHttpRequest` is that one can't undo `setRequestHeader`.

Once the header is set, it's set. Additional calls add information to the header, don't overwrite it.

For instance:

```
xhr.setRequestHeader('X-Auth', '123');
xhr.setRequestHeader('X-Auth', '456');

// the header will be:
// X-Auth: 123, 456
```

getResponseHeader (name)

Gets the response header with the given `name` (except `Set-Cookie` and `Set-Cookie2`).

For instance:

```
xhr.getResponseHeader('Content-Type')
```

getAllResponseHeaders()

Returns all response headers, except `Set-Cookie` and `Set-Cookie2`.

Headers are returned as a single line, e.g.:

```
Cache-Control: max-age=31536000  
Content-Length: 4260  
Content-Type: image/png  
Date: Sat, 08 Sep 2012 16:53:16 GMT
```

The line break between headers is always `"\r\n"` (doesn't depend on OS), so we can easily split it into individual headers. The separator between the name and the value is always a colon followed by a space `": "`. That's fixed in the specification.

So, if we want to get an object with name/value pairs, we need to throw in a bit JS.

Like this (assuming that if two headers have the same name, then the latter one overwrites the former one):

```
let headers = xhr  
  .getAllResponseHeaders()  
  .split('\r\n')  
  .reduce((result, current) => {  
    let [name, value] = current.split(': ');  
    result[name] = value;  
    return result;  
  }, {});
```

POST, FormData

To make a POST request, we can use the built-in [FormData](#)  object.

The syntax:

```
let formData = new FormData([form]); // creates an object, optionally fill from <fo
formData.append(name, value); // appends a field
```

We create it, optionally from a form, `append` more fields if needed, and then:

1. `xhr.open('POST', ...)` – use `POST` method.
2. `xhr.send(formData)` to submit the form to the server.

For instance:

```
<form name="person">
  <input name="name" value="John">
  <input name="surname" value="Smith">
</form>

<script>
  // pre-fill FormData from the form
  let formData = new FormData(document.forms.person);

  // add one more field
  formData.append("middle", "Lee");

  // send it out
  let xhr = new XMLHttpRequest();
  xhr.open("POST", "/article/xmlhttprequest/post/user");
  xhr.send(formData);

</script>
```

The form is sent with `multipart/form-data` encoding.

Or, if we like JSON more, then `JSON.stringify` and send as a string.

Just don't forget to set the header `Content-Type: application/json`, many server-side frameworks automatically decode JSON with it:

```
let xhr = new XMLHttpRequest();

let json = JSON.stringify({
  name: "John",
  surname: "Smith"
});

xhr.open("POST", '/submit')
xhr.setRequestHeader('Content-type', 'application/json; charset=utf-8');

xhr.send(json);
```

The `.send(body)` method is pretty omnivore. It can send almost everything, including `Blob` and `BufferSource` objects.

Upload progress

The `progress` event only works on the downloading stage.

That is: if we `POST` something, `XMLHttpRequest` first uploads our data (the request body), then downloads the response.

If we're uploading something big, then we're surely more interested in tracking the upload progress. But `xhr.onprogress` doesn't help here.

There's another object `xhr.upload`, without methods, exclusively for upload events.

The event list is similar to `xhr` events, but `xhr.upload` triggers them on uploading:

- `loadstart` – upload started.
- `progress` – triggers periodically during the upload.
- `abort` – upload aborted.
- `error` – non-HTTP error.
- `load` – upload finished successfully.
- `timeout` – upload timed out (if `timeout` property is set).
- `loadend` – upload finished with either success or error.

Example of handlers:

```
xhr.upload.onprogress = function(event) {
  alert(`Uploaded ${event.loaded} of ${event.total} bytes`);
};

xhr.upload.onload = function() {
  alert(`Upload finished successfully.`);
};

xhr.upload.onerror = function() {
  alert(`Error during the upload: ${xhr.status}`);
};
```

Here's a real-life example: file upload with progress indication:

```
<input type="file" onchange="upload(this.files[0])">
```

```

<script>
function upload(file) {
  let xhr = new XMLHttpRequest();

  // track upload progress
  xhr.upload.onprogress = function(event) {
    console.log(`Uploaded ${event.loaded} of ${event.total}`);
  };

  // track completion: both successful or not
  xhr.onloadend = function() {
    if (xhr.status == 200) {
      console.log("success");
    } else {
      console.log("error " + this.status);
    }
  };

  xhr.open("POST", "/article/xmlhttprequest/post/upload");
  xhr.send(file);
}
</script>

```

Cross-origin requests

`XMLHttpRequest` can make cross-domain requests, using the same CORS policy as `fetch`.

Just like `fetch`, it doesn't send cookies and HTTP-authorization to another origin by default. To enable them, set `xhr.withCredentials` to `true`:

```

let xhr = new XMLHttpRequest();
xhr.withCredentials = true;

xhr.open('POST', 'http://anywhere.com/request');
...

```

See the chapter [Fetch: Cross-Origin Requests](#) for details about cross-origin headers.

Summary

Typical code of the GET-request with `XMLHttpRequest`:

```

let xhr = new XMLHttpRequest();

xhr.open('GET', '/my/url');

```

```

xhr.send();

xhr.onload = function() {
  if (xhr.status !== 200) { // HTTP error?
    // handle error
    alert( 'Error: ' + xhr.status);
    return;
  }

  // get the response from xhr.response
};

xhr.onprogress = function(event) {
  // report progress
  alert(`Loaded ${event.loaded} of ${event.total}`);
};

xhr.onerror = function() {
  // handle non-HTTP error (e.g. network down)
};

```

There are actually more events, the [modern specification](#) lists them (in the lifecycle order):

- `loadstart` – the request has started.
- `progress` – a data packet of the response has arrived, the whole response body at the moment is in `responseText`.
- `abort` – the request was canceled by the call `xhr.abort()`.
- `error` – connection error has occurred, e.g. wrong domain name. Doesn't happen for HTTP-errors like 404.
- `load` – the request has finished successfully.
- `timeout` – the request was canceled due to timeout (only happens if it was set).
- `loadend` – triggers after `load`, `error`, `timeout` or `abort`.

The `error`, `abort`, `timeout`, and `load` events are mutually exclusive. Only one of them may happen.

The most used events are load completion (`load`), load failure (`error`), or we can use a single `loadend` handler and check the response to see what happened.

We've already seen another event: `readystatechange`. Historically, it appeared long ago, before the specification settled. Nowadays, there's no need to use it, we can replace it with newer events, but it can often be found in older scripts.

If we need to track uploading specifically, then we should listen to same events on `xhr.upload` object.

Resumable file upload

With `fetch` method it's fairly easy to upload a file.

How to resume the upload after lost connection? There's no built-in option for that, but we have the pieces to implement it.

Resumable uploads should come with upload progress indication, as we expect big files (if we may need to resume). So, as `fetch` doesn't allow to track upload progress, we'll use [XMLHttpRequest](#).

Not-so-useful progress event

To resume upload, we need to know how much was uploaded till the connection was lost.

There's `xhr.upload.onprogress` to track upload progress.

Unfortunately, it's useless here, as it triggers when the data is *sent*, but was it received by the server? The browser doesn't know.

Maybe it was buffered by a local network proxy, or maybe the remote server process just died and couldn't process them, or it was just lost in the middle when the connection broke, and didn't reach the receiver.

So, this event is only useful to show a nice progress bar.

To resume upload, we need to know exactly the number of bytes received by the server. And only the server can tell that.

Algorithm

1. First, we create a file id, to uniquely identify the file we're uploading, e.g.

```
let fileId = file.name + '-' + file.size + '-' + file.lastModifiedDate;
```

That's needed for resume upload, to tell the server what we're resuming.

2. Send a request to the server, asking how many bytes it already has, like this:

```
let response = await fetch('status', {
  headers: {
    'X-File-Id': fileId
  }
});
```

```
// The server has that many bytes
let startByte = +await response.text();
```

This assumes that the server tracks file uploads by `X-File-Id` header. Should be implemented at server-side.

3. Then, we can use `Blob` method `slice` to send the file from `startByte`:

```
xhr.open("POST", "upload", true);

// send file id, so that the server knows which file to resume
xhr.setRequestHeader('X-File-Id', fileId);
// send the byte we're resuming from, so the server knows we're resuming
xhr.setRequestHeader('X-Start-Byte', startByte);

xhr.upload.onprogress = (e) => {
  console.log(`Uploaded ${startByte + e.loaded} of ${startByte + e.total}`);
};

// file can be from input.files[0] or another source
xhr.send(file.slice(startByte));
```

Here we send the server both file id as `X-File-Id`, so it knows which file we're uploading, and the starting byte as `X-Start-Byte`, so it knows we're not uploading it initially, but resuming.

The server should check its records, and if there was an upload of that file, and the current uploaded size is exactly `X-Start-Byte`, then append the data to it.

Here's the demo with both client and server code, written on Node.js.

It works only partially on this site, as Node.js is behind another server named Nginx, that buffers uploads, passing them to Node.js when fully complete.

But you can download it and run locally for the full demonstration:

<https://plnkr.co/edit/uwIHsRek1zB1NhjxDjC9?p=preview> ↗

As you can see, modern networking methods are close to file managers in their capabilities – control over headers, progress indicator, sending file parts, etc.

Long polling

Long polling is the simplest way of having persistent connection with server, that doesn't use any specific protocol like WebSocket or Server Side Events.

Being very easy to implement, it's also good enough in a lot of cases.

Regular Polling

The simplest way to get new information from the server is polling.

That is, periodical requests to the server: “Hello, I’m here, do you have any information for me?”. For example, once in 10 seconds.

In response, the server first takes a notice to itself that the client is online, and second – sends a packet of messages it got till that moment.

That works, but there are downsides:

1. Messages are passed with a delay up to 10 seconds.
2. Even if there are no messages, the server is bombed with requests every 10 seconds. That’s quite a load to handle for backend, speaking performance-wise.

So, if we’re talking about a very small service, the approach may be viable.

But generally, it needs an improvement.

Long polling

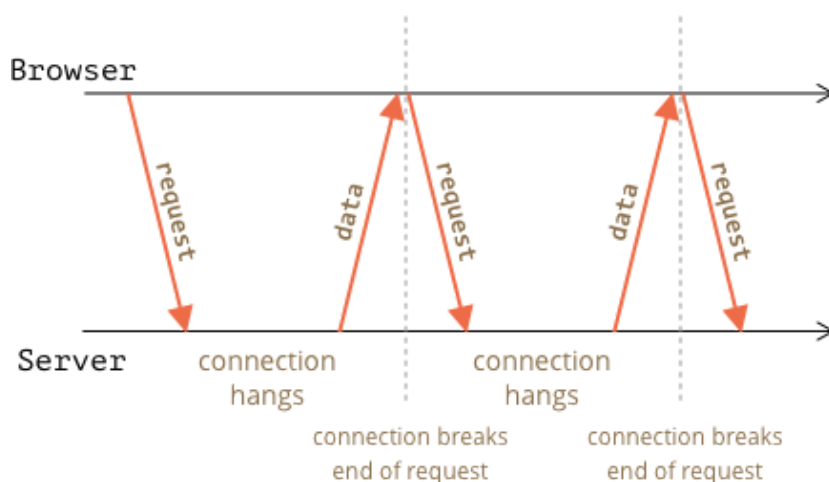
Long polling – is a better way to poll the server.

It’s also very easy to implement, and delivers messages without delays.

The flow:

1. A request is sent to the server.
2. The server doesn’t close the connection until it has a message.
3. When a message appears – the server responds to the request with the data.
4. The browser makes a new request immediately.

The situation when the browser sent a request and has a pending connection with the server, is standard for this method. Only when a message is delivered, the connection is reestablished.



Even if the connection is lost, because of, say, a network error, the browser immediately sends a new request.

A sketch of client-side code:

```
async function subscribe() {
  let response = await fetch("/subscribe");

  if (response.status == 502) {
    // Connection timeout, happens when the connection was pending for too long
    // let's reconnect
    await subscribe();
  } else if (response.status != 200) {
    // Show Error
    showMessage(response.statusText);
    // Reconnect in one second
    await new Promise(resolve => setTimeout(resolve, 1000));
    await subscribe();
  } else {
    // Got message
    let message = await response.text();
    showMessage(message);
    await subscribe();
  }
}

subscribe();
```

The `subscribe()` function makes a fetch, then waits for the response, handles it and calls itself again.

Server should be ok with many pending connections

The server architecture must be able to work with many pending connections.

Certain server architectures run a process per connect. For many connections there will be as many processes, and each process takes a lot of memory. So many connections just consume it all.

That's often the case for backends written in PHP, Ruby languages, but technically isn't a language, but rather implementation issue.

Backends written using Node.js usually don't have such problems.

Demo: a chat

Here's a demo:

<https://plnkr.co/edit/p0VXeg5MlwpxPjq7LWdR?p=preview> 

Area of usage

Long polling works great in situations when messages are rare.

If messages come very often, then the chart of requesting-receiving messages, painted above, becomes saw-like.

Every message is a separate request, supplied with headers, authentication overhead, and so on.

So, in this case, another method is preferred, such as [WebSocket](#) or [Server Sent Events](#).

WebSocket

The `WebSocket` protocol, described in the specification [RFC 6455](#) [↗](#) provides a way to exchange data between browser and server via a persistent connection.

Once a websocket connection is established, both client and server may send the data to each other.

WebSocket is especially great for services that require continuous data exchange, e.g. online games, real-time trading systems and so on.

A simple example

To open a websocket connection, we need to create `new WebSocket` using the special protocol `ws` in the url:

```
let socket = new WebSocket("ws://javascript.info");
```

There's also encrypted `wss://` protocol. It's like HTTPS for websockets.

Always prefer `wss://`

The `wss://` protocol not only encrypted, but also more reliable.

That's because `ws://` data is not encrypted, visible for any intermediary. Old proxy servers do not know about WebSocket, they may see "strange" headers and abort the connection.

On the other hand, `wss://` is WebSocket over TLS, (same as HTTPS is HTTP over TLS), the transport security layer encrypts the data at sender and decrypts at the receiver, so it passes encrypted through proxies. They can't see what's inside and let it through.

Once the socket is created, we should listen to events on it. There are totally 4 events:

- **open** – connection established,
- **message** – data received,
- **error** – websocket error,
- **close** – connection closed.

...And if we'd like to send something, then `socket.send(data)` will do that.

Here's an example:

```
let socket = new WebSocket("wss://javascript.info/article/websocket/demo/hello");

socket.onopen = function(e) {
  alert("[open] Connection established, send -> server");
  socket.send("My name is John");
};

socket.onmessage = function(event) {
  alert(`[message] Data received: ${event.data} <- server`);
};

socket.onclose = function(event) {
  if (event.wasClean) {
    alert(`[close] Connection closed cleanly, code=${event.code} reason=${event.reason}`);
  } else {
    // e.g. server process killed or network down
    // event.code is usually 1006 in this case
    alert('[close] Connection died');
  }
};

socket.onerror = function(error) {
  alert(`[error] ${error.message}`);
};
```

For demo purposes, there's a small server [server.js](#) written in Node.js, for the example above, running. It responds with "hello", then waits 5 seconds and closes the connection.

So you'll see events `open` → `message` → `close`.

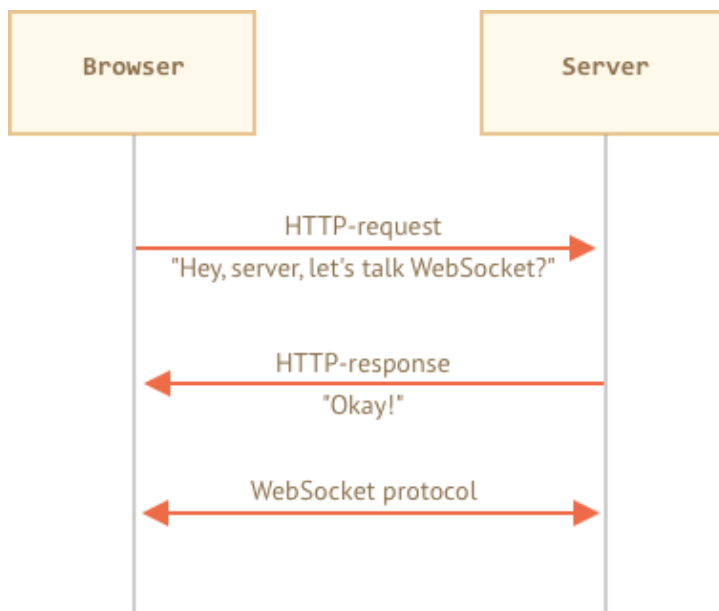
That's actually it, we can talk WebSocket already. Quite simple, isn't it?

Now let's talk more in-depth.

Opening a websocket

When `new WebSocket(url)` is created, it starts connecting immediately.

During the connection the browser (using headers) asks the server: “Do you support WebSocket?” And if the server replies “yes”, then the talk continues in WebSocket protocol, which is not HTTP at all.



Here's an example of browser request for `new WebSocket("wss://javascript.info/chat")`.

```
GET /chat
Host: javascript.info
Origin: https://javascript.info
Connection: Upgrade
Upgrade: websocket
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
```

- **Origin** – the origin of the client page, e.g. `https://javascript.info`. WebSocket objects are cross-origin by nature. There are no special headers or other limitations. Old servers are unable to handle WebSocket anyway, so there are no compatibility issues. But **Origin** header is important, as it allows the server to decide whether or not to talk WebSocket with this website.
- **Connection: Upgrade** – signals that the client would like to change the protocol.
- **Upgrade: websocket** – the requested protocol is “websocket”.
- **Sec-WebSocket-Key** – a random browser-generated key for security.
- **Sec-WebSocket-Version** – WebSocket protocol version, 13 is the current one.

WebSocket handshake can't be emulated

We can't use `XMLHttpRequest` or `fetch` to make this kind of HTTP-request, because JavaScript is not allowed to set these headers.

If the server agrees to switch to WebSocket, it should send code 101 response:

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzE0TBULZA1C2g=
```

Here `Sec-WebSocket-Accept` is `Sec-WebSocket-Key`, recoded using a special algorithm. The browser uses it to make sure that the response corresponds to the request.

Afterwards, the data is transferred using WebSocket protocol, we'll see its structure ("frames") soon. And that's not HTTP at all.

Extensions and subprotocols

There may be additional headers `Sec-WebSocket-Extensions` and `Sec-WebSocket-Protocol` that describe extensions and subprotocols.

For instance:

- `Sec-WebSocket-Extensions: deflate-frame` means that the browser supports data compression. An extension is something related to transferring the data, not data itself.
- `Sec-WebSocket-Protocol: soap, wamp` means that we'd like to transfer not just any data, but the data in [SOAP](#) or WAMP ("The WebSocket Application Messaging Protocol") protocols. WebSocket subprotocols are registered in the [IANA catalogue](#).

`Sec-WebSocket-Extensions` header is sent by the browser automatically, with a list of possible extensions it supports.

`Sec-WebSocket-Protocol` header depends on us: we decide what kind of data we send. The second optional parameter of `new WebSocket` is just for that, it lists subprotocols:

```
let socket = new WebSocket("wss://javascript.info/chat", ["soap", "wamp"]);
```

The server should respond with a list of protocols and extensions that it agrees to use.

For example, the request:

```
GET /chat
Host: javascript.info
Upgrade: websocket
Connection: Upgrade
Origin: https://javascript.info
Sec-WebSocket-Key: Iv8io/9s+lYFgZWcXczP8Q==
Sec-WebSocket-Version: 13
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap, wamp
```

Response:

```
101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: hsBlbuDTkk24srzE0TBULZAlC2g=
Sec-WebSocket-Extensions: deflate-frame
Sec-WebSocket-Protocol: soap
```

Here the server responds that it supports the extension “deflate-frame”, and only SOAP of the requested subprotocols.

WebSocket data

WebSocket communication consists of “frames” – data fragments, that can be sent from either side, and can be of several kinds:

- “text frames” – contain text data that parties send to each other.
- “binary data frames” – contain binary data that parties send to each other.
- “ping/pong frames” are used to check the connection, sent from the server, the browser responds to these automatically.
- “connection close frame” and a few other service frames.

In the browser, we directly work only with text or binary frames.

WebSocket .send() method can send either text or binary data.

A call `socket.send(body)` allows `body` in string or a binary format, including `Blob`, `ArrayBuffer`, etc. No settings required: just send it out.

When we receive the data, text always comes as string. And for binary data, we can choose between `Blob` and `ArrayBuffer` formats.

The `socket.bufferType` is `"blob"` by default, so binary data comes in Blobs.

`Blob` is a high-level binary object, it directly integrates with `<a>`, `` and other tags, so that's a sane default. But for binary processing, to access individual data bytes, we can change it to `"arraybuffer"`:

```
socket.bufferType = "arraybuffer";
socket.onmessage = (event) => {
  // event.data is either a string (if text) or arraybuffer (if binary)
};
```

Rate limiting

Imagine, our app is generating a lot of data to send. But the user has a slow network connection, maybe on a mobile, outside of a city.

We can call `socket.send(data)` again and again. But the data will be buffered (stored) in memory and sent out only as fast as network speed allows.

The `socket.bufferedAmount` property stores how many bytes are buffered at this moment, waiting to be sent over the network.

We can examine it to see whether the socket is actually available for transmission.

```
// every 100ms examine the socket and send more data
// only if all the existing data was sent out
setInterval(() => {
  if (socket.bufferedAmount == 0) {
    socket.send(moreData());
  }
}, 100);
```

Connection close

Normally, when a party wants to close the connection (both browser and server have equal rights), they send a “connection close frame” with a numeric code and a textual reason.

The method for that is:

```
socket.close([code], [reason]);
```

- `code` is a special WebSocket closing code (optional)
- `reason` is a string that describes the reason of closing (optional)

Then the other party in `close` event handler gets the code and the reason, e.g.:


```
// closing party:
socket.close(1000, "Work complete");

// the other party
socket.onclose = event => {
  // event.code === 1000
  // event.reason === "Work complete"
  // event.wasClean === true (clean close)
};
```

Most common code values:

- **1000** – the default, normal closure (used if no **code** supplied),
- **1006** – no way to such code manually, indicates that the connection was lost (no close frame).

There are other codes like:

- **1001** – the party is going away, e.g. server is shutting down, or a browser leaves the page,
- **1009** – the message is too big to process,
- **1011** – unexpected error on server,
- ...and so on.

Please refer to the [RFC6455, §7.4.1](#) for the full list.

WebSocket codes are somewhat like HTTP codes, but different. In particular, any codes less than **1000** are reserved, there'll be an error if we try to set such a code.

```
// in case connection is broken
socket.onclose = event => {
  // event.code === 1006
  // event.reason === ""
  // event.wasClean === false (no closing frame)
};
```

Connection state

To get connection state, additionally there's **socket.readyState** property with values:

- **0** – “CONNECTING”: the connection has not yet been established,
- **1** – “OPEN”: communicating,
- **2** – “CLOSING”: the connection is closing,

- **3** – “CLOSED”: the connection is closed.

Chat example

Let's review a chat example using browser WebSocket API and Node.js WebSocket module <https://github.com/websockets/ws> .

HTML: there's a `<form>` to send messages and a `<div>` for incoming messages:

```
<!-- message form -->
<form name="publish">
  <input type="text" name="message">
  <input type="submit" value="Send">
</form>

<!-- div with messages -->
<div id="messages"></div>
```

JavaScript is also simple. We open a socket, then on form submission – `socket.send(message)` , on incoming message – append it to `div#messages` :

```
let socket = new WebSocket("wss://javascript.info/article/websocket/chat/ws");

// send message from the form
document.forms.publish.onsubmit = function() {
  let outgoingMessage = this.message.value;

  socket.send(outgoingMessage);
  return false;
};

// show message in div#messages
socket.onmessage = function(event) {
  let message = event.data;

  let messageElem = document.createElement('div');
  messageElem.textContent = message;
  document.getElementById('messages').prepend(messageElem);
}
```

Server-side code is a little bit beyond our scope here. We're using browser WebSocket API, a server may have another library.

Still it can also be pretty simple. We'll use Node.js with <https://github.com/websockets/ws> module for websockets.

The server-side algorithm will be:

1. Create `clients = new Set()` – a set of sockets.
2. For each accepted websocket, `clients.add(socket)` and add `message` event listener for its messages.
3. When a message received: iterate over clients and send it to everyone.
4. When a connection is closed: `clients.delete(socket)`.

```
const ws = new require('ws');
const wss = new ws.Server({noServer: true});

const clients = new Set();

http.createServer((req, res) => {
  // here we only handle websocket connections
  // in real project we'd have some other code herre to handle non-websocket requests
  wss.handleUpgrade(req, req.socket, Buffer.alloc(0), onSocketConnect);
});

function onSocketConnect(ws) {
  clients.add(ws);

  ws.on('message', function(message) {
    message = message.slice(0, 50); // max message length will be 50

    for(let client of clients) {
      client.send(message);
    }
  });

  ws.on('close', function() {
    clients.delete(ws);
  });
}
```

Here's the working example:

Send

You can also download it (upper-right button in the iframe) and run locally. Just don't forget to install [Node.js](#) and `npm install ws` before running.

Summary

WebSocket is a modern way to have persistent browser-server connections.

- WebSockets don't have cross-origin limitations.
- They are well-supported in browsers.
- Can send/receive strings and binary data.

The API is simple.

Methods:

- `socket.send(data)`,
- `socket.close([code], [reason])`.

Events:


- `open`,
- `message`,
- `error`,
- `close`.

WebSocket by itself does not include reconnection, authentication and many other high-level mechanisms. So there are client/server libraries for that, and it's also possible to implement these capabilities manually.

Sometimes, to integrate WebSocket into existing project, people run WebSocket server in parallel with the main HTTP-server, and they share a single database. Requests to WebSocket use `wss://ws.site.com`, a subdomain that leads to WebSocket server, while `https://site.com` goes to the main HTTP-server.

Surely, other ways of integration are also possible. Many servers (such as Node.js) can support both HTTP and WebSocket protocols.

Server Sent Events

The [Server-Sent Events](#)  specification describes a built-in class `EventSource`, that keeps connection with the server and allows to receive events from it.

Similar to `WebSocket`, the connection is persistent.

But there are several important differences:

WebSocket	EventSource
Bi-directional: both client and server can exchange messages	One-directional: only server sends data
Binary and text data	Only text
WebSocket protocol	Regular HTTP

`EventSource` is a less-powerful way of communicating with the server than `WebSocket` .

Why should one ever use it?

The main reason: it's simpler. In many applications, the power of `WebSocket` is a little bit too much.

We need to receive a stream of data from server: maybe chat messages or market prices, or whatever. That's what `EventSource` is good at. Also it supports auto-reconnect, something we need to implement manually with `WebSocket` . Besides, it's a plain old HTTP, not a new protocol.

Getting messages

To start receiving messages, we just need to create `new EventSource(url)` .

The browser will connect to `url` and keep the connection open, waiting for events.

The server should respond with status 200 and the header `Content-Type: text/event-stream` , then keep the connection and write messages into it in the special format, like this:

```
data: Message 1

data: Message 2

data: Message 3
data: of two lines
```

- A message text goes after `data:` , the space after the semicolon is optional.
- Messages are delimited with double line breaks `\n\n` .
- To send a line break `\n` , we can immediately one more `data:` (3rd message above).

In practice, complex messages are usually sent JSON-encoded, so line-breaks are encoded within them.

For instance:

```
data: {"user":"John","message":"First line\nSecond line"}
```

...So we can assume that one `data:` holds exactly one message.

For each such message, the `message` event is generated:

```
let eventSource = new EventSource("/events/subscribe");

eventSource.onmessage = function(event) {
  console.log("New message", event.data);
  // will log 3 times for the data stream above
};

// or eventSource.addEventListener('message', ...)
```

Cross-domain requests

`EventSource` supports cross-origin requests, like `fetch` any other networking methods. We can use any URL:

```
let source = new EventSource("https://another-site.com/events");
```

The remote server will get the `Origin` header and must respond with `Access-Control-Allow-Origin` to proceed.

To pass credentials, we should set the additional option `withCredentials`, like this:

```
let source = new EventSource("https://another-site.com/events", {
  withCredentials: true
});
```

Please see the chapter [Fetch: Cross-Origin Requests](#) for more details about cross-domain headers.

Reconnection

Upon creation, `new EventSource` connects to the server, and if the connection is broken – reconnects.

That's very convenient, as we don't have to care about it.

There's a small delay between reconnections, a few seconds by default.

The server can set the recommended delay using `retry:` in response (in milliseconds):

```
retry: 15000
data: Hello, I set the reconnection delay to 15 seconds
```

The `retry` may come both together with some data, or as a standalone message.

The browser should wait that much before reconnect. If the network connection is lost, the browser may wait till it's restored, and then retry.

- If the server wants the browser to stop reconnecting, it should respond with HTTP status 204.
- If the browser wants to close the connection, it should call `eventSource.close()`:

```
let eventSource = new EventSource(...);  
  
eventSource.close();
```

Also, there will be no reconnection if the response has an incorrect `Content-Type` or its HTTP status differs from 301, 307, 200 and 204. The connection the `"error"` event is emitted, and the browser won't reconnect.

i Please note:

There's no way to "reopen" a closed connection. If we'd like to connect again, just create a new `EventSource`.

Message id

When a connection breaks due to network problems, either side can't be sure which messages were received, and which weren't.

To correctly resume the connection, each message should have an `id` field, like this:

```
data: Message 1  
id: 1  
  
data: Message 2  
id: 2  
  
data: Message 3  
data: of two lines  
id: 3
```

When a message with `id` is received, the browser:

- Sets the property `eventSource.lastEventId` to its value.

- Upon reconnection sends the header `Last-Event-ID` with that `id`, so that the server may re-send following messages.

i Put `id`: after data:

Please note: the `id` is appended below the message data, to ensure that `lastEventId` is updated after the message data is received.

Connection status: `readyState`

The `EventSource` object has `readyState` property, that has one of three values:

```
EventSource.CONNECTING = 0; // connecting or reconnecting
EventSource.OPEN = 1;      // connected
EventSource.CLOSED = 2;    // connection closed
```

When an object is created, or the connection is down, it's always `EventSource.CONNECTING` (equals `0`).

We can query this property to know the state of `EventSource`.

Event types

By default `EventSource` object generates three events:

- `message` – a message received, available as `event.data`.
- `open` – the connection is open.
- `error` – the connection could not be established, e.g. the server returned HTTP 500 status.

The server may specify another type of event with `event: ...` at the event start.

For example:

```
event: join
data: Bob

data: Hello

event: leave
data: Bob
```


To handle custom events, we must use `addEventListener` , not `onmessage` :

```
eventSource.addEventListener('join', event => {
  alert(`Joined ${event.data}`);
});

eventSource.addEventListener('message', event => {
  alert(`Said: ${event.data}`);
});

eventSource.addEventListener('leave', event => {
  alert(`Left ${event.data}`);
});
```

Full example

Here's the server that sends messages with `1` , `2` , `3` , then `bye` and breaks the connection.

Then the browser automatically reconnects.

<https://plnkr.co/edit/LmOdPFHJdD3ylrRGhkSF?p=preview> ↗

Summary

The `EventSource` object communicates with the server. It establishes a persistent connection and allows the server to send messages over it.

It offers:

- Automatic reconnect, with tunable `retry` timeout.
- Message ids to resume events, the last identifier is sent in `Last-Event-ID` header.
- The current state is in the `readyState` property.

That makes `EventSource` a viable alternative to `WebSocket` , as it's more low-level and lacks these features.

In many real-life applications, the power of `EventSource` is just enough.

Supported in all modern browsers (not IE).

The syntax is:

```
let source = new EventSource(url, [credentials]);
```

The second argument has only one possible option: `{ withCredentials: true }`, it allows sending cross-domain credentials.

Overall cross-domain security is same as for `fetch` and other network methods.

Properties of an `EventSource` object

`readyState`

The current connection state: either `EventSource.CONNECTING (=0)`, `EventSource.OPEN (=1)` or `EventSource.CLOSED (=2)`.

`lastEventId`

The last received `id`. Upon reconnection the browser sends it in the header `Last-Event-ID`.

Methods

`close()`

Closes the connection `соединение`.

Events

`message`

Message received, the data is in `event.data`.

`open`

The connection is established.

`error`

In case of an error, including both lost connection (will auto-reconnect) and fatal errors. We can check `readyState` to see if the reconnection is being attempted.

The server may set a custom event name in `event: .` Such events should be handled using `addEventListener`, not `on<event>`.

Server response format

The server sends messages, delimited by `\n\n`.


Message parts may start with:

- `data:` – message body, a sequence of multiple `data` is interpreted as a single message, with `\n` between the parts.
- `id:` – renews `lastEventId`, sent in `Last-Event-ID` on reconnect.
- `retry:` – recommends a retry delay for reconnections in ms. There's no way to set it from JavaScript.

- `event :` – even name, must precede `data :` .

Storing data in the browser

Cookies, `document.cookie`

Cookies are small strings of data that are stored directly in the browser. They are a part of HTTP protocol, defined by [RFC 6265](#)  specification.

Cookies are usually set by a web-server using response `Set-Cookie` HTTP-header. Then the browser automatically adds them to (almost) every request to the same domain using `Cookie` HTTP-header.

One of the most widespread use cases is authentication:

1. Upon sign in, the server uses `Set-Cookie` HTTP-header in the response to set a cookie with a unique “session identifier”.
2. Next time when the request is set to the same domain, the browser sends the over the net using `Cookie` HTTP-header.
3. So the server knows who made the request.

We can also access cookies from the browser, using `document.cookie` property.

There are many tricky things about cookies and their options. In this chapter we'll cover them in detail.

Reading from `document.cookie`

Assuming you're on a website, it's possible to see the cookies from it, like this:

```
// At javascript.info, we use Google Analytics for statistics,  
// so there should be some cookies  
alert( document.cookie ); // cookie1=value1; cookie2=value2;...
```

The value of `document.cookie` consists of `name=value` pairs, delimited by `;` . Each one is a separate cookie.

To find a particular cookie, we can split `document.cookie` by `;` , and then find the right name. We can use either a regular expression or array functions to do that.

We leave it as an exercise for the reader. Also, at the end of the chapter you'll find helper functions to manipulate cookies.

Writing to `document.cookie`

We can write to `document.cookie`. But it's not a data property, it's an accessor. An assignment to it is treated specially.

A write operation to `document.cookie` passes through the browser that updates cookies mentioned in it, but doesn't touch other cookies.

For instance, this call sets a cookie with the name `user` and value `John`:

```
document.cookie = "user=John"; // update only cookie named 'user'
alert(document.cookie); // show all cookies
```

If you run it, then probably you'll see multiple cookies. That's because `document.cookie=` operation does not overwrite all cookies. It only sets the mentioned cookie `user`.

Technically, name and value can have any characters, but to keep the formatting valid they should be escaped using a built-in `encodeURIComponent` function:

```
// special values, need encoding
let name = "my name";
let value = "John Smith"

// encodes the cookie as my%20name=John%20Smith
document.cookie = encodeURIComponent(name) + '=' + encodeURIComponent(value);

alert(document.cookie); // ...; my%20name=John%20Smith
```

Limitations

There are few limitations:

- The `name=value` pair, after `encodeURIComponent`, should not exceed 4kb. So we can't store anything huge in a cookie.
- The total number of cookies per domain is limited to around 20+, the exact limit depends on a browser.

Cookies have several options, many of them are important and should be set.

The options are listed after `key=value`, delimited by `;`, like this:

```
document.cookie = "user=John; path=/; expires=Tue, 19 Jan 2038 03:14:07 GMT"
```

path

- `path=/mypath`

The url path prefix, the cookie will be accessible for pages under that path. Must be absolute. By default, it's the current path.

If a cookie is set with `path=/admin`, it's visible at pages `/admin` and `/admin/something`, but not at `/home` or `/adminpage`.

Usually, we should set `path` to the root: `path=/' to make the cookie accessible from all website pages.`

domain

- `domain=site.com`

A domain where the cookie is accessible. In practice though, there are limitations. We can't set any domain.

By default, a cookie is accessible only at the domain that set it. So, if the cookie was set by `site.com`, we won't get it `other.com`.

...But what's more tricky, we also won't get the cookie at a subdomain `forum.site.com`!

```
// at site.com
document.cookie = "user=John"

// at forum.site.com
alert(document.cookie); // no user
```

There's no way to let a cookie be accessible from another 2nd-level domain, so `other.com` will never receive a cookie set at `site.com`.

It's a safety restriction, to allow us to store sensitive data in cookies, that should be available only on one site.

...But if we'd like to allow subdomains like `forum.site.com` get a cookie, that's possible. When setting a cookie at `site.com`, we should explicitly set `domain` option to the root domain: `domain=site.com`:

```
// at site.com
// make the cookie accessible on any subdomain *.site.com:
document.cookie = "user=John; domain=site.com"

// later
```

```
// at forum.site.com
alert(document.cookie); // has cookie user=John
```

For historical reasons, `domain=.site.com` (a dot before `site.com`) also works the same way, allowing access to the cookie from subdomains. That's an old notation, should be used if we need to support very old browsers.

So, `domain` option allows to make a cookie accessible at subdomains.

expires, max-age

By default, if a cookie doesn't have one of these options, it disappears when the browser is closed. Such cookies are called "session cookies"

To let cookies survive browser close, we can set either `expires` or `max-age` option.

- **`expires=Tue, 19 Jan 2038 03:14:07 GMT`**

Cookie expiration date, when the browser will delete it automatically.

The date must be exactly in this format, in GMT timezone. We can use `date.toUTCString` to get it. For instance, we can set the cookie to expire in 1 day:

```
// +1 day from now
let date = new Date(Date.now() + 86400e3);
date = date.toUTCString();
document.cookie = "user=John; expires=" + date;
```

If we set `expires` to a date in the past, the cookie is deleted.

- **`max-age=3600`**

An alternative to `expires`, specifies the cookie expiration in seconds from the current moment.

If zero or negative, then the cookie is deleted:

```
// cookie will die +1 hour from now
document.cookie = "user=John; max-age=3600";

// delete cookie (let it expire right now)
document.cookie = "user=John; max-age=0";
```

secure

- **secure**

The cookie should be transferred only over HTTPS.

By default, if we set a cookie at `http://site.com`, then it also appears at `https://site.com` and vice versa.

That is, cookies are domain-based, they do not distinguish between the protocols.

With this option, if a cookie is set by `https://site.com`, then it doesn't appear when the same site is accessed by HTTP, as `http://site.com`. So if a cookie has sensitive content that should never be sent over unencrypted HTTP, then the flag is the right thing.

```
// assuming we're on https:// now
// set the cookie secure (only accessible if over HTTPS)
document.cookie = "user=John; secure";
```

samesite

That's another security attribute `samesite`. It's designed to protect from so-called XSRF (cross-site request forgery) attacks.

To understand how it works and when it's useful, let's take a look at XSRF attacks.

XSRF attack

Imagine, you are logged into the site `bank.com`. That is: you have an authentication cookie from that site. Your browser sends it to `bank.com` with every request, so that it recognizes you and performs all sensitive financial operations.

Now, while browsing the web in another window, you occasionally come to another site `evil.com`, that automatically submits a form `<form action="https://bank.com/pay">` to `bank.com` with input fields that initiate a transaction to the hacker's account.

The form is submitted from `evil.com` directly to the bank site, and your cookie is also sent, just because it's sent every time you visit `bank.com`. So the bank recognizes you and actually performs the payment.



That's called a cross-site request forgery (or XSRF) attack.

Real banks are protected from it of course. All forms generated by `bank.com` have a special field, so called "xsrf protection token", that an evil page can't neither generate, nor somehow extract from a remote page (it can submit a form there, but can't get the data back).

But that takes time to implement: we need to ensure that every form has the token field, and we must also check all requests.

Enter cookie `samesite` option

The cookie `samesite` option provides another way to protect from such attacks, that (in theory) should not require "xsrf protection tokens".

It has two possible values:

- **`samesite=strict` (same as `samesite` without value)**

A cookie with `samesite=strict` is never sent if the user comes from outside the site.

In other words, whether a user follows a link from their mail or submits a form from `evil.com`, or does any operation that originates from another domain, the cookie is not sent.

If authentication cookies have `samesite` option, then XSRF attack has no chances to succeed, because a submission from `evil.com` comes without cookies. So `bank.com` will not recognize the user and will not proceed with the payment.

The protection is quite reliable. Only operations that come from `bank.com` will send the `samesite` cookie.

Although, there's a small inconvenience.

When a user follows a legitimate link to `bank.com`, like from their own notes, they'll be surprised that `bank.com` does not recognize them. Indeed, `samesite=strict` cookies are not sent in that case.

We could work around that by using two cookies: one for "general recognition", only for the purposes of saying: "Hello, John", and the other one for data-changing operations with `samesite=strict`. Then a person coming from outside of the site will see a welcome, but payments must be initiated from the bank website.

- **`samesite=lax`**

A more relaxed approach that also protects from XSRF and doesn't break user experience.

Lax mode, just like `strict`, forbids the browser to send cookies when coming from outside the site, but adds an exception.

A `samesite=lax` cookie is sent if both of these conditions are true:

1. The HTTP method is “safe” (e.g. GET, but not POST).

The full list of safe HTTP methods is in the [RFC7231 specification](#) . Basically, these are the methods that should be used for reading, but not writing the data. They must not perform any data-changing operations. Following a link is always GET, the safe method.

2. The operation performs top-level navigation (changes URL in the browser address bar).

That’s usually true, but if the navigation is performed in an `<iframe>` , then it’s not top-level. Also, AJAX requests do not perform any navigation, hence they don’t fit.

So, what `samesite=lax` does is basically allows a most common “go to URL” operation to have cookies. E.g. opening a website link from notes satisfies these conditions.

But anything more complicated, like AJAX request from another site or a form submission loses cookies.

If that’s fine for you, then adding `samesite=lax` will probably not break the user experience and add protection.

Overall, `samesite` is a great option, but it has an important drawback:

- `samesite` is ignored (not supported) by old browsers, year 2017 or so.

So if we solely rely on `samesite` to provide protection, then old browsers will be vulnerable.

But we surely can use `samesite` together with other protection measures, like xsrf tokens, to add an additional layer of defence and then, in the future, when old browsers die out, we’ll probably be able to drop xsrf tokens.

httpOnly

This option has nothing to do with JavaScript, but we have to mention it for completeness.

The web-server uses `Set -Cookie` header to set a cookie. And it may set the `httpOnly` option.

This option forbids any JavaScript access to the cookie. We can’t see such cookie or manipulate it using `document.cookie` .

That’s used as a precaution measure, to protect from certain attacks when a hacker injects his own JavaScript code into a page and waits for a user to visit that page.

That shouldn't be possible at all, a hacker should not be able to inject their code into our site, but there may be bugs that let hackers do it.

Normally, if such thing happens, and a user visits a web-page with hacker's code, then that code executes and gains access to `document.cookie` with user cookies containing authentication information. That's bad.

But if a cookie is `httpOnly`, then `document.cookie` doesn't see it, so it is protected.

Appendix: Cookie functions

Here's a small set of functions to work with cookies, more convenient than a manual modification of `document.cookie`.

There exist many cookie libraries for that, so these are for demo purposes. Fully working though.

getCookie(name)

The shortest way to access cookie is to use a [regular expression](#).

The function `getCookie(name)` returns the cookie with the given `name`:

```
// returns the cookie with the given name,  
// or undefined if not found  
function getCookie(name) {  
  let matches = document.cookie.match(new RegExp(  
    "(?:^|; )" + name.replace(/[\\.$?*|{}\\(\\)\\[\\]\\^\\+\\/]/g, '\\\\$1') + "=([^\"]*)" )  
  ));  
  return matches ? decodeURIComponent(matches[1]) : undefined;  
}
```

Here `new RegExp` is generated dynamically, to match `; name=<value>`.

Please note that a cookie value is encoded, so `getCookie` uses a built-in `decodeURIComponent` function to decode it.

setCookie(name, value, options)

Sets the cookie `name` to the given `value` with `path=/` by default (can be modified to add other defaults):

```
function setCookie(name, value, options = {}) {  
  
  options = {  
    path: '/',  
    // add other defaults here if necessary  
    ...options  
  };  
};
```

```

if (options.expires.toUTCString) {
    options.expires = options.expires.toUTCString();
}

let updatedCookie = encodeURIComponent(name) + "=" + encodeURIComponent(value);

for (let optionKey in options) {
    updatedCookie += "; " + optionKey;
    let optionValue = options[optionKey];
    if (optionValue !== true) {
        updatedCookie += "=" + optionValue;
    }
}

document.cookie = updatedCookie;
}

// Example of use:
setCookie('user', 'John', {secure: true, 'max-age': 3600});

```

deleteCookie(name)

To delete a cookie, we can call it with a negative expiration date:

```

function deleteCookie(name) {
    setCookie(name, "", {
        'max-age': -1
    })
}

```

Updating or deleting must use same path and domain

Please note: when we update or delete a cookie, we should use exactly the same path and domain options as when we set it.

Together: [cookie.js](#).

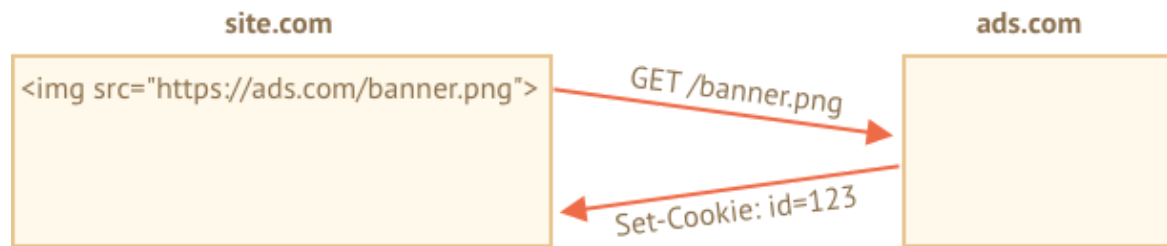
Appendix: Third-party cookies

A cookie is called “third-party” if it’s placed by domain other than the user is visiting.

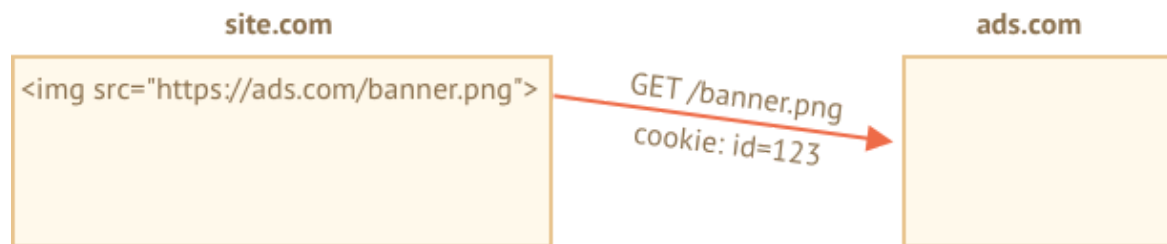
For instance:

1. A page at `site.com` loads a banner from another site: ``.
2. Along with the banner, the remote server at `ads.com` may set `Set-Cookie` header with cookie like `id=1234`. Such cookie originates from `ads.com`

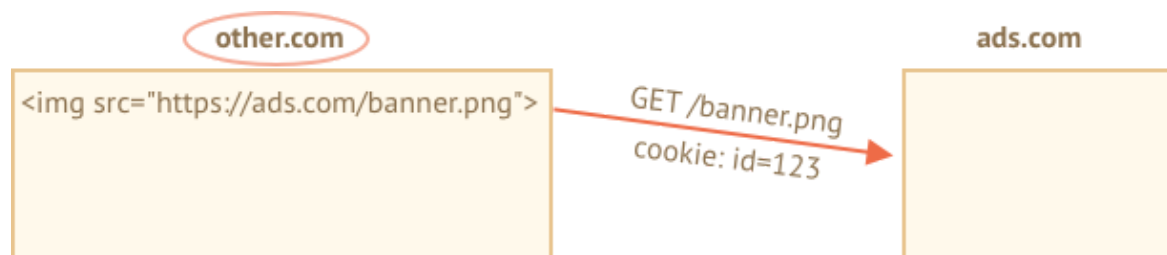
domain, and will only be visible at `ads.com` :



3. Next time when `ads.com` is accessed, the remote server gets the `id` cookie and recognizes the user:



4. What's even more important, when the users moves from `site.com` to another site `other.com` that also has a banner, then `ads.com` gets the cookie, as it belongs to `ads.com` , thus recognizing the visitor and tracking him as he moves between sites:



Third-party cookies are traditionally used for tracking and ads services, due to their nature. They are bound to the originating domain, so `ads.com` can track the same user between different sites, if they all access it.

Naturally, some people don't like being tracked, so browsers allow to disable such cookies.

Also, some modern browsers employ special policies for such cookies:

- Safari does not allow third-party cookies at all.
- Firefox comes with a "black list" of third-party domains where it blocks third-party cookies.

i Please note:

If we load a script from a third-party domain, like `<script src="https://google-analytics.com/analytics.js">`, and that script uses `document.cookie` to set a cookie, then such cookie is not third-party.

If a script sets a cookie, then no matter where the script came from – it belongs to the domain of the current webpage.

Appendix: GDPR

This topic is not related to JavaScript at all, just something to keep in mind when setting cookies.

There's a legislation in Europe called GDPR, that enforces a set of rules for websites to respect users' privacy. And one of such rules is to require an explicit permission for tracking cookies from a user.

Please note, that's only about tracking/identifying cookies.

So, if we set a cookie that just saves some information, but neither tracks nor identifies the user, then we are free to do it.

But if we are going to set a cookie with an authentication session or a tracking id, then a user must allow that.

Websites generally have two variants of following GDPR. You must have seen them both already in the web:

1. If a website wants to set tracking cookies only for authenticated users.

To do so, the registration form should have a checkbox like "accept the privacy policy", the user must check it, and then the website is free to set auth cookies.

2. If a website wants to set tracking cookies for everyone.

To do so legally, a website shows a modal "splash screen" for newcomers, and require them to agree for cookies. Then the website can set them and let people see the content. That can be disturbing for new visitors though. No one likes to see "must-click" modal splash screens instead of the content. But GDPR requires an explicit agreement.

GDPR is not only about cookies, it's about other privacy-related issues too, but that's too much beyond our scope.

Summary

`document.cookie` provides access to cookies

- write operations modify only cookies mentioned in it.
- name/value must be encoded.
- one cookie up to 4kb, 20+ cookies per site (depends on a browser).

Cookie options:

- `path=/` , by default current path, makes the cookie visible only under that path.
- `domain=site.com` , by default a cookie is visible on current domain only, if set explicitly to the domain, makes the cookie visible on subdomains.
- `expires` or `max-age` sets cookie expiration time, without them the cookie dies when the browser is closed.
- `secure` makes the cookie HTTPS-only.
- `samesite` forbids the browser to send the cookie with requests coming from outside the site, helps to prevent XSRF attacks.

Additionally:

- Third-party cookies may be forbidden by the browser, e.g. Safari does that by default.
- When setting a tracking cookie for EU citizens, GDPR requires to ask for permission.

LocalStorage, sessionStorage

Web storage objects `localStorage` and `sessionStorage` allow to save key/value pairs in the browser.

What's interesting about them is that the data survives a page refresh (for `sessionStorage`) and even a full browser restart (for `localStorage`). We'll see that very soon.

We already have cookies. Why additional objects?

- Unlike cookies, web storage objects are not sent to server with each request. Because of that, we can store much more. Most browsers allow at least 2 megabytes of data (or more) and have settings to configure that.
- Also unlike cookies, the server can't manipulate storage objects via HTTP headers. Everything's done in JavaScript.
- The storage is bound to the origin (domain/protocol/port triplet). That is, different protocols or subdomains infer different storage objects, they can't access data from each other.

Both storage objects provide same methods and properties:

- `setItem(key, value)` – store key/value pair.

- `getItem(key)` – get the value by key.
- `removeItem(key)` – remove the key with its value.
- `clear()` – delete everything.
- `key(index)` – get the key on a given position.
- `length` – the number of stored items.

As you can see, it's like a `Map` collection (`setItem/getItem/removeItem`), but also keeps elements order and allows to access by index with `key(index)`.

Let's see how it works.

localStorage demo

The main features of `localStorage` are:

- Shared between all tabs and windows from the same origin.
- The data does not expire. It remains after the browser restart and even OS reboot.

For instance, if you run this code...

```
localStorage.setItem('test', 1);
```

...And close/open the browser or just open the same page in a different window, then you can get it like this:

```
alert( localStorage.getItem('test') ); // 1
```

We only have to be on the same origin (domain/port/protocol), the url path can be different.

The `localStorage` is shared between all windows with the same origin, so if we set the data in one window, the change becomes visible in another one.

Object-like access

We can also use a plain object way of getting/setting keys, like this:

```
// set key
localStorage.test = 2;

// get key
alert( localStorage.test ); // 2
```

```
// remove key
delete localStorage.test;
```

That's allowed for historical reasons, and mostly works, but generally not recommended for two reasons:

1. If the key is user-generated, it can be anything, like `length` or `toString`, or another built-in method of `localStorage`. In that case `getItem/setItem` work fine, while object-like access fails:

```
let key = 'length';
localStorage[key] = 5; // Error, can't assign length
```

2. There's a `storage` event, it triggers when we modify the data. That event does not happen for object-like access. We'll see that later in this chapter.

Looping over keys

As we've seen, the methods provide “get/set/remove by key” functionality. But how to get all saved values or keys?

Unfortunately, storage objects are not iterable.

One way is to loop over them as over an array:

```
for(let i=0; i<localStorage.length; i++) {
  let key = localStorage.key(i);
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

Another way is to use `for key in localStorage` loop, just as we do with regular objects.

It iterates over keys, but also outputs few built-in fields that we don't need:

```
// bad try
for(let key in localStorage) {
  alert(key); // shows getItem, setItem and other built-in stuff
}
```

...So we need either to filter fields from the prototype with `hasOwnProperty` check:


```
for(let key in localStorage) {
  if (!localStorage.hasOwnProperty(key)) {
    continue; // skip keys like "setItem", "getItem" etc
  }
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

...Or just get the “own” keys with `Object.keys` and then loop over them if needed:

```
let keys = Object.keys(localStorage);
for(let key of keys) {
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

The latter works, because `Object.keys` only returns the keys that belong to the object, ignoring the prototype.

Strings only

Please note that both key and value must be strings.

If were any other type, like a number, or an object, it gets converted to string automatically:

```
sessionStorage.user = {name: "John"};
alert(sessionStorage.user); // [object Object]
```

We can use `JSON` to store objects though:

```
sessionStorage.user = JSON.stringify({name: "John"});

// sometime later
let user = JSON.parse( sessionStorage.user );
alert( user.name ); // John
```

Also it is possible to stringify the whole storage object, e.g. for debugging purposes:

```
// added formatting options to JSON.stringify to make the object look nicer
alert( JSON.stringify(localStorage, null, 2) );
```

sessionStorage

The `sessionStorage` object is used much less often than `localStorage`.

Properties and methods are the same, but it's much more limited:

- The `sessionStorage` exists only within the current browser tab.
 - Another tab with the same page will have a different storage.
 - But it is shared between iframes in the tab (assuming they come from the same origin).
- The data survives page refresh, but not closing/opening the tab.

Let's see that in action.

Run this code...

```
sessionStorage.setItem('test', 1);
```

...Then refresh the page. Now you can still get the data:

```
alert( sessionStorage.getItem('test') ); // after refresh: 1
```

...But if you open the same page in another tab, and try again there, the code above returns `null`, meaning "nothing found".

That's exactly because `sessionStorage` is bound not only to the origin, but also to the browser tab. For that reason, `sessionStorage` is used sparingly.

Storage event

When the data gets updated in `localStorage` or `sessionStorage`, [storage](#) event triggers, with properties:

- `key` – the key that was changed (`null` if `.clear()` is called).
- `oldValue` – the old value (`null` if the key is newly added).
- `newValue` – the new value (`null` if the key is removed).
- `url` – the url of the document where the update happened.
- `storageArea` – either `localStorage` or `sessionStorage` object where the update happened.

The important thing is: the event triggers on all `window` objects where the storage is accessible, except the one that caused it.

Let's elaborate.

Imagine, you have two windows with the same site in each. So `localStorage` is shared between them.

If both windows are listening for `window.onstorage`, then each one will react on updates that happened in the other one.

```
// triggers on updates made to the same storage from other documents
window.onstorage = event => {
  if (event.key !== 'now') return;
  alert(event.key + ':' + event.newValue + " at " + event.url);
};

localStorage.setItem('now', Date.now());
```

Please note that the event also contains: `event.url` – the url of the document where the data was updated.

Also, `event.storageArea` contains the storage object – the event is the same for both `sessionStorage` and `localStorage`, so `storageArea` references the one that was modified. We may even want to set something back in it, to “respond” to a change.

That allows different windows from the same origin to exchange messages.

Modern browsers also support [Broadcast channel API](#), the special API for same-origin inter-window communication, it's more full featured, but less supported. There are libraries that polyfill that API, based on `localStorage`, that make it available everywhere.

Summary

Web storage objects `localStorage` and `sessionStorage` allow to store key/value in the browser.

- Both `key` and `value` must be strings.
- The limit is 2mb+, depends on the browser.
- They do not expire.
- The data is bound to the origin (domain/port/protocol).

localStorage	sessionStorage
Shared between all tabs and windows with the same origin	Visible within a browser tab, including iframes from the same origin
Survives browser restart	Survives page refresh (but not tab close)

API:

- `setItem(key, value)` – store key/value pair.
- `getItem(key)` – get the value by key.
- `removeItem(key)` – remove the key with its value.
- `clear()` – delete everything.
- `key(index)` – get the key number `index`.
- `length` – the number of stored items.
- Use `Object.keys` to get all keys.
- We access keys as object properties, in that case `storage` event isn't triggered.

Storage event:

- Triggers on `setItem`, `removeItem`, `clear` calls.
- Contains all the data about the operation, the document `url` and the storage object.
- Triggers on all `window` objects that have access to the storage except the one that generated it (within a tab for `sessionStorage`, globally for `localStorage`).

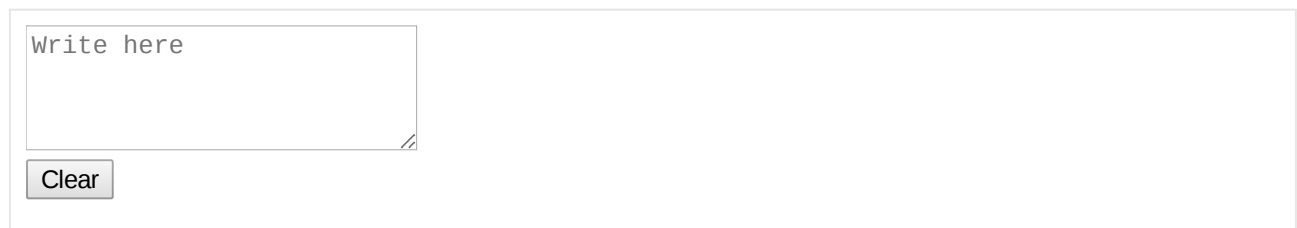
✓ Tasks

Autosave a form field

Create a `textarea` field that “autosaves” its value on every change.

So, if the user occasionally closes the page, and opens it again, he'll find his unfinished input at place.

Like this:



[Open a sandbox for the task.](#) ↗

[To solution](#)

IndexedDB

IndexedDB is a built-in database, much more powerful than `localStorage`.

- Key/value storage: value can be (almost) anything, multiple key types.
- Supports transactions for reliability.
- Supports key range queries, indexes.
- Can store much more data than `localStorage`.

That power is usually excessive for traditional client-server apps. IndexedDB is intended for offline apps, to be combined with ServiceWorkers and other technologies.

The native interface to IndexedDB, described in the specification <https://www.w3.org/TR/IndexedDB>, is event-based.

We can also use `async/await` with the help of a promise-based wrapper, like <https://github.com/jakearchibald/idb>. That's pretty convenient, but the wrapper is not perfect, it can't replace events for all cases. So we'll start with events, and then, after we gain understanding of IndexedDb, we'll use the wrapper.

Open database

To start working with IndexedDB, we first need to open a database.

The syntax:

```
let openRequest = indexedDB.open(name, version);
```

- `name` – a string, the database name.
- `version` – a positive integer version, by default `1` (explained below).

We can have many databases with different names, but all of them exist within the current origin (domain/protocol/port). Different websites can't access databases of each other.

After the call, we need to listen to events on `openRequest` object:

- `success`: database is ready, there's the "database object" in `openRequest.result`, that we should use it for further calls.
- `error`: opening failed.
- `upgradeneeded`: database is ready, but its version is outdated (see below).

IndexedDB has a built-in mechanism of “schema versioning”, absent in server-side databases.

Unlike server-side databases, IndexedDB is client-side, the data is stored in the browser, so we, developers, don't have direct access to it. But when we publish a new version of our app, we may need to update the database.

If the local database version is less than specified in `open`, then a special event `upgradeneeded` is triggered, and we can compare versions and upgrade data structures as needed.

The event also triggers when the database did not exist yet, so we can perform initialization.

When we first publish our app, we open it with version `1` and perform the initialization in `upgradeneeded` handler:

```
let openRequest = indexedDB.open("store", 1);

openRequest.onupgradeneeded = function() {
  // triggers if the client had no database
  // ...perform initialization...
};

openRequest.onerror = function() {
  console.error("Error", openRequest.error);
};

openRequest.onsuccess = function() {
  let db = openRequest.result;
  // continue to work with database using db object
};
```

When we publish the 2nd version:

```
let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = function() {
  // the existing database version is less than 2 (or it doesn't exist)
  let db = openRequest.result;
  switch(db.version) { // existing db version
    case 0:
      // version 0 means that the client had no database
      // perform initialization
    case 1:
      // client had version 1
      // update
  }
};
```

So, in `openRequest.onupgradeneeded` we update the database. Soon we'll see how it's done. And then, only if its handler finishes without errors, `openRequest.onsuccess` triggers.

After `openRequest.onsuccess` we have the database object in `openRequest.result`, that we'll use for further operations.

To delete a database:

```
let deleteRequest = indexedDB.deleteDatabase(name)
// deleteRequest.onsuccess/onerror tracks the result
```



Can we open an old version?

Now what if we try to open a database with a lower version than the current one? E.g. the existing DB version is 3, and we try to `open(...2)`.

That's an error, `openRequest.onerror` triggers.

Such thing may happen if the visitor loaded an outdated code, e.g. from a proxy cache. We should check `db.version`, suggest him to reload the page. And also re-check our caching headers to ensure that the visitor never gets old code.

Parallel update problem

As we're talking about versioning, let's tackle a small related problem.

Let's say, a visitor opened our site in a browser tab, with database version 1.

Then we rolled out an update, and the same visitor opens our site in another tab. So there are two tabs, both with our site, but one has an open connection with DB version 1, while the other one attempts to update it in `upgradeneeded` handler.

The problem is that a database is shared between two tabs, as that's the same site, same origin. And it can't be both version 1 and 2. To perform the update to version 2, all connections to version 1 must be closed.

In order to organize that, the `versionchange` event triggers an open database object when a parallel upgrade is attempted. We should listen to it, so that we should close the database (and probably suggest the visitor to reload the page, to load the updated code).

If we don't close it, then the second, new connection will be blocked with `blocked` event instead of `success`.

Here's the code to do that:

```

let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = ...;
openRequest.onerror = ...;

openRequest.onsuccess = function() {
  let db = openRequest.result;

  db.onversionchange = function() {
    db.close();
    alert("Database is outdated, please reload the page.")
  };

  // ...the db is ready, use it...
};

openRequest.onblocked = function() {
  // there's another open connection to same database
  // and it wasn't closed after db.onversionchange triggered for them
};

```

Here we do two things:

1. Add `db.onversionchange` listener after a successful opening, to be informed about a parallel update attempt.
2. Add `openRequest.onblocked` listener to handle the case when an old connection wasn't closed. This doesn't happen if we close it in `db.onversionchange`.

There are other variants. For example, we can take time to close things gracefully in `db.onversionchange`, prompt the visitor to save the data before the connection is closed. The new updating connection will be blocked immediately after `db.onversionchange` finished without closing, and we can ask the visitor in the new tab to close other tabs for the update.

Such update collision happens rarely, but we should at least have some handling for it, e.g. `onblocked` handler, so that our script doesn't surprise the user by dying silently.

Object store

To store something in IndexedDB, we need an *object store*.

An object store is a core concept of IndexedDB. Counterparts in other databases are called “tables” or “collections”. It's where the data is stored. A database may have multiple stores: one for users, another one for goods, etc.

Despite being named an “object store”, primitives can be stored too.

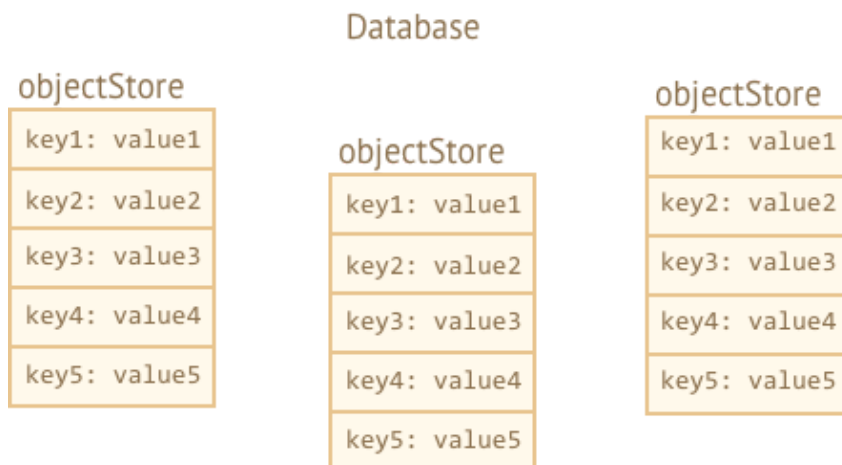
We can store almost any value, including complex objects.

IndexedDB uses the [standard serialization algorithm](#) to clone-and-store an object. It's like `JSON.stringify`, but more powerful, capable of storing much more datatypes.

An example of object that can't be stored: an object with circular references. Such objects are not serializable. `JSON.stringify` also fails for such objects.

There must be a unique key for every value in the store.

A key must have a type one of: number, date, string, binary, or array. It's an unique identifier: we can search/remove/update values by the key.



As we'll see very soon, we can provide a key when we add a value to the store, similar to `localStorage`. But when we store objects, IndexedDB allows to setup an object property as the key, that's much more convenient. Or we can auto-generate keys.

But we need to create an object store first.

The syntax to create an object store:

```
db.createObjectStore(name[, keyOptions]);
```

Please note, the operation is synchronous, no `await` needed.

- `name` is the store name, e.g. `"books"` for books,
- `keyOptions` is an optional object with one of two properties:
 - `keyPath` – a path to an object property that IndexedDB will use as the key, e.g. `id`.
 - `autoIncrement` – if `true`, then the key for a newly stored object is generated automatically, as an ever-incrementing number.

If we don't supply `keyOptions`, then we'll need to provide a key explicitly later, when storing an object.

For instance, this object store uses `id` property as the key:

```
db.createObjectStore('books', {keyPath: 'id'});
```

An object store can only be created/modified while updating the DB version, in `upgradeneeded` handler.

That's a technical limitation. Outside of the handler we'll be able to add/remove/update the data, but object stores can be created/removed/alterd only during version update.

To perform database version upgrade, there are two main approaches:

1. We can implement per-version upgrade functions: from 1 to 2, from 2 to 3, from 3 to 4 etc. Then, in `upgradeneeded` we can compare versions (e.g. old 2, now 4) and run per-version upgrades step by step, for every intermediate version (2 to 3, then 3 to 4).
2. Or we can just examine the database: get a list of existing object stores as `db.objectStoreNames`. That object is a [DOMStringList](#) that provides `contains(name)` method to check for existence. And then we can do updates depending on what exists and what doesn't.

For small databases the second variant may be simpler.

Here's the demo of the second approach:

```
let openRequest = indexedDB.open("db", 2);

// create/upgrade the database without version checks
openRequest.onupgradeneeded = function() {
  let db = openRequest.result;
  if (!db.objectStoreNames.contains('books')) { // if there's no "books" store
    db.createObjectStore('books', {keyPath: 'id'}); // create it
  }
};
```

To delete an object store:

```
db.deleteObjectStore('books')
```

Transactions

The term “transaction” is generic, used in many kinds of databases.

A transaction is a group operations, that should either all succeed or all fail.

For instance, when a person buys something, we need:

1. Subtract the money from their account.
2. Add the item to their inventory.

It would be pretty bad if we complete the 1st operation, and then something goes wrong, e.g. lights out, and we fail to do the 2nd. Both should either succeed (purchase complete, good!) or both fail (at least the person kept their money, so they can retry).

Transactions can guarantee that.

All data operations must be made within a transaction in IndexedDB.

To start a transaction:

```
db.transaction(store[, type]);
```

- `store` is a store name that the transaction is going to access, e.g. `"books"`. Can be an array of store names if we're going to access multiple stores.
- `type` – a transaction type, one of:
 - `readonly` – can only read, the default.
 - `readwrite` – can only read and write the data, but not create/remove/alter object stores.

There's also `versionchange` transaction type: such transactions can do everything, but we can't create them manually. IndexedDB automatically creates a `versionchange` transaction when opening the database, for `updateneeded` handler. That's why it's a single place where we can update the database structure, create/remove object stores.

Why there exist different types of transactions?

Performance is the reason why transactions need to be labeled either `readonly` and `readwrite`.

Many `readonly` transactions are able to access concurrently the same store, but `readwrite` transactions can't. A `readwrite` transaction “locks” the store for writing. The next transaction must wait before the previous one finishes before accessing the same store.

After the transaction is created, we can add an item to the store, like this:

```

let transaction = db.transaction("books", "readwrite"); // (1)

// get an object store to operate on it
let books = transaction.objectStore("books"); // (2)

let book = {
  id: 'js',
  price: 10,
  created: new Date()
};

let request = books.add(book); // (3)

request.onsuccess = function() { // (4)
  console.log("Book added to the store", request.result);
};

request.onerror = function() {
  console.log("Error", request.error);
};

```

There were basically four steps:

1. Create a transaction, mention all stores it's going to access, at (1) .
2. Get the store object using `transaction.objectStore(name)` , at (2) .
3. Perform the request to the object store `books.add(book)` , at (3) .
4. ...Handle request success/error (4) , then we can make other requests if needed, etc.

Object stores support two methods to store a value:

- **put(value, [key])** Add the `value` to the store. The `key` is supplied only if the object store did not have `keyPath` or `autoIncrement` option. If there's already a value with same key, it will be replaced.
- **add(value, [key])** Same as `put` , but if there's already a value with the same key, then the request fails, and an error with the name `"ConstraintError"` is generated.

Similar to opening a database, we can send a request: `books.add(book)` , and then wait for `success/error` events.

- The `request.result` for `add` is the key of the new object.
- The error is in `request.error` (if any).

Transactions' autocommit

In the example above we started the transaction and made `add` request. But as we stated previously, a transaction may have multiple associated requests, that must either all success or all fail. How do we mark the transaction as finished, no more requests to come?

The short answer is: we don't.

In the next version 3.0 of the specification, there will probably be a manual way to finish the transaction, but right now in 2.0 there isn't.

When all transaction requests are finished, and the `microtasks queue` is empty, it is committed automatically.

Usually, we can assume that a transaction commits when all its requests are complete, and the current code finishes.

So, in the example above no special call is needed to finish the transaction.

Transactions auto-commit principle has an important side effect. We can't insert an async operation like `fetch`, `setTimeout` in the middle of transaction. IndexedDB will not keep the transaction waiting till these are done.

In the code below `request2` in line `(*)` fails, because the transaction is already committed, can't make any request in it:

```
let request1 = books.add(book);

request1.onsuccess = function() {
  fetch('/').then(response => {
    let request2 = books.add(anotherBook); // (*)
    request2.onerror = function() {
      console.log(request2.error.name); // TransactionInactiveError
    };
  });
};
```

That's because `fetch` is an asynchronous operation, a macrotask. Transactions are closed before the browser starts doing macrotasks.

Authors of IndexedDB spec believe that transactions should be short-lived. Mostly for performance reasons.

Notably, `readwrite` transactions "lock" the stores for writing. So if one part of application initiated `readwrite` on `books` object store, then another part that wants to do the same has to wait: the new transaction "hangs" till the first one is done. That can lead to strange delays if transactions take a long time.

So, what to do?

In the example above we could make a new `db.transaction` right before the new request `(*)`.

But it will be even better, if we'd like to keep the operations together, in one transaction, to split apart IndexedDB transactions and “other” async stuff.

First, make `fetch`, prepare the data if needed, afterwards create a transaction and perform all the database requests, it'll work then.

To detect the moment of successful completion, we can listen to `transaction.oncomplete` event:

```
let transaction = db.transaction("books", "readwrite");

// ...perform operations...

transaction.oncomplete = function() {
  console.log("Transaction is complete");
};
```

Only `complete` guarantees that the transaction is saved as a whole. Individual requests may succeed, but the final write operation may go wrong (e.g. I/O error or something).

To manually abort the transaction, call:

```
transaction.abort();
```

That cancels all modification made by the requests in it and triggers `transaction.onabort` event.

Error handling

Write requests may fail.

That's to be expected, not only because of possible errors at our side, but also for reasons not related to the transaction itself. For instance, the storage quota may be exceeded. So we must be ready to handle such case.

A failed request automatically aborts the transaction, canceling all its changes.

In some situations, we may want to handle the failure (e.g. try another request), without canceling existing changes, and continue the transaction. That's possible. The `request.onerror` handler is able to prevent the transaction abort by calling `event.preventDefault()`.

In the example below a new book is added with the same key (`id`) as the existing one. The `store.add` method generates a `"ConstraintError"` in that case. We handle it without canceling the transaction:

```
let transaction = db.transaction("books", "readwrite");

let book = { id: 'js', price: 10 };

let request = transaction.objectStore("books").add(book);

request.onerror = function(event) {
  // ConstraintError occurs when an object with the same id already exists
  if (request.error.name === "ConstraintError") {
    console.log("Book with such id already exists"); // handle the error
    event.preventDefault(); // don't abort the transaction
    // use another key for the book?
  } else {
    // unexpected error, can't handle it
    // the transaction will abort
  }
};

transaction.onabort = function() {
  console.log("Error", transaction.error);
};
```

Event delegation

Do we need `onerror`/`onsuccess` for every request? Not every time. We can use event delegation instead.

IndexedDB events bubble: `request` → `transaction` → `database`.

All events are DOM events, with capturing and bubbling, but usually only bubbling stage is used.

So we can catch all errors using `db.onerror` handler, for reporting or other purposes:

```
db.onerror = function(event) {
  let request = event.target; // the request that caused the error

  console.log("Error", request.error);
};
```

...But what if an error is fully handled? We don't want to report it in that case.

We can stop the bubbling and hence `db.onerror` by using `event.stopPropagation()` in `request.onerror`.

```
request.onerror = function(event) {
  if (request.error.name === "ConstraintError") {
    console.log("Book with such id already exists"); // handle the error
    event.preventDefault(); // don't abort the transaction
    event.stopPropagation(); // don't bubble error up, "chew" it
  } else {
    // do nothing
    // transaction will be aborted
    // we can take care of error in transaction.onabort
  }
};
```

Searching by keys

There are two main types of search in an object store:

1. By a key or a key range. That is: by `book.id` in our “books” storage.
2. By another object field, e.g. `book.price`.

First let's deal with the keys and key ranges (1).

Methods that involve searching support either exact keys or so-called “range queries” – [IDBKeyRange](#) objects that specify a “key range”.

Ranges are created using following calls:

- `IDBKeyRange.lowerBound(lower, [open])` means: `>lower` (or `≥lower` if `open` is true)
- `IDBKeyRange.upperBound(upper, [open])` means: `<upper` (or `≤upper` if `open` is true)
- `IDBKeyRange.bound(lower, upper, [lowerOpen], [upperOpen])` means: between `lower` and `upper`, with optional equality if the corresponding `open` is true.
- `IDBKeyRange.only(key)` – a range that consists of only one `key`, rarely used.

All searching methods accept a `query` argument that can be either an exact key or a key range:

- `store.get(query)` – search for the first value by a key or a range.
- `store.getAll([query], [count])` – search for all values, limit by `count` if given.
- `store.getKey(query)` – search for the first key that satisfies the query, usually a range.

- `store.getAllKeys([query], [count])` – search for all keys that satisfy the query, usually a range, up to `count` if given.
- `store.count([query])` – get the total count of keys that satisfy the query, usually a range.

For instance, we have a lot of books in our store. Remember, the `id` field is the key, so all these methods can search by `id`.

Request examples:

```
// get one book
books.get('js')

// get books with 'css' < id < 'html'
books.getAll(IDBKeyRange.bound('css', 'html'))

// get books with 'html' <= id
books.getAll(IDBKeyRange.lowerBound('html', true))

// get all books
books.getAll()

// get all keys: id >= 'js'
books.getAllKeys(IDBKeyRange.lowerBound('js', true))
```

Object store is always sorted

Object store sorts values by key internally.

So requests that return many values always return them in sorted by key order.

Searching by any field with an index

To search by other object fields, we need to create an additional data structure named “index”.

An index is an “add-on” to the store that tracks a given object field. For each value of that field, it stores a list of keys for objects that have that value. There will be a more detailed picture below.

The syntax:

```
objectStore.createIndex(name, keyPath, [options]);
```

- **name** – index name,

- **keyPath** – path to the object field that the index should track (we're going to search by that field),
- **option** – an optional object with properties:
 - **unique** – if true, then there may be only one object in the store with the given value at the **keyPath**. The index will enforce that by generating an error if we try to add a duplicate.
 - **multiEntry** – only used if the value on **keyPath** is an array. In that case, by default, the index will treat the whole array as the key. But if **multiEntry** is true, then the index will keep a list of store objects for each value in that array. So array members become index keys.

In our example, we store books keyed by **id**.

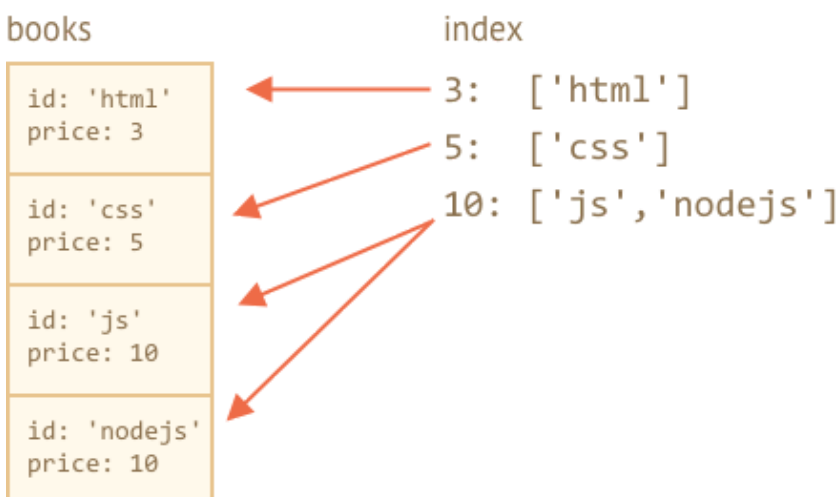
Let's say we want to search by **price**.

First, we need to create an index. It must be done in **upgradeneeded**, just like an object store:

```
openRequest.onupgradeneeded = function() {
  // we must create the index here, in versionchange transaction
  let books = db.createObjectStore('books', {keyPath: 'id'});
  let index = inventory.createIndex('price_idx', 'price');
};
```

- The index will track **price** field.
- The price is not unique, there may be multiple books with the same price, so we don't set **unique** option.
- The price is not an array, so **multiEntry** flag is not applicable.

Imagine that our **inventory** has 4 books. Here's the picture that shows exactly what the **index** is:



As said, the index for each value of `price` (second argument) keeps the list of keys that have that price.

The index keeps itself up to date automatically, we don't have to care about it.

Now, when we want to search for a given price, we simply apply the same search methods to the index:

```
let transaction = db.transaction("books"); // readonly
let books = transaction.objectStore("books");
let priceIndex = books.index("price_idx");

let request = priceIndex.getAll(10);

request.onsuccess = function() {
  if (request.result !== undefined) {
    console.log("Books", request.result); // array of books with price=10
  } else {
    console.log("No such books");
  }
};
```

We can also use `IDBKeyRange` to create ranges and looks for cheap/expensive books:

```
// find books where price < 5
let request = priceIndex.getAll(IDBKeyRange.upperBound(5));
```

Indexes are internally sorted by the tracked object field, `price` in our case. So when we do the search, the results are also sorted by `price`.

Deleting from store

The `delete` method looks up values to delete by a query, the call format is similar to `getAll`:

- **`delete(query)`** – delete matching values by query.

For instance:

```
// delete the book with id='js'
books.delete('js');
```

If we'd like to delete books based on a price or another object field, then we should first find the key in the index, and then call `delete`:

```
// find the key where price = 5
let request = priceIndex.getKey(5);

request.onsuccess = function() {
  let id = request.result;
  let deleteRequest = books.delete(id);
};
```

To delete everything:

```
books.clear(); // clear the storage.
```

Cursors

Methods like `getAll/getAllKeys` return an array of keys/values.

But an object storage can be huge, bigger than the available memory. Then `getAll` will fail to get all records as an array.

What to do?

Cursors provide the means to work around that.

A *cursor* is a special object that traverses the object storage, given a query, and returns one key/value at a time, thus saving memory.

As an object store is sorted internally by key, a cursor walks the store in key order (ascending by default).

The syntax:

```
// like getAll, but with a cursor:
let request = store.openCursor(query, [direction]);

// to get keys, not values (like getAllKeys): store.openKeyCursor
```

- **query** is a key or a key range, same as for `getAll`.
- **direction** is an optional argument, which order to use:
 - "next" – the default, the cursor walks up from the record with the lowest key.
 - "prev" – the reverse order: down from the record with the biggest key.

- "nextunique", "prevunique" – same as above, but skip records with the same key (only for cursors over indexes, e.g. for multiple books with price=5 only the first one will be returned).

The main difference of the cursor is that `request.onsuccess` triggers multiple times: once for each result.

Here's an example of how to use a cursor:

```
let transaction = db.transaction("books");
let books = transaction.objectStore("books");

let request = books.openCursor();

// called for each book found by the cursor
request.onsuccess = function() {
  let cursor = request.result;
  if (cursor) {
    let key = cursor.key; // book key (id field)
    let value = cursor.value; // book object
    console.log(key, value);
    cursor.continue();
  } else {
    console.log("No more books");
  }
};
```

The main cursor methods are:

- `advance(count)` – advance the cursor `count` times, skipping values.
- `continue([key])` – advance the cursor to the next value in range matching (or immediately after `key` if given).

Whether there are more values matching the cursor or not – `onsuccess` gets called, and then in `result` we can get the cursor pointing to the next record, or `undefined`.

In the example above the cursor was made for the object store.

But we also can make a cursor over an index. As we remember, indexes allow to search by an object field. Cursors over indexes to precisely the same as over object stores – they save memory by returning one value at a time.

For cursors over indexes, `cursor.key` is the index key (e.g. price), and we should use `cursor.primaryKey` property the object key:

```
let request = priceIdx.openCursor(IDBKeyRange.upperBound(5));
```

```
// called for each record
request.onsuccess = function() {
  let cursor = request.result;
  if (cursor) {
    let key = cursor.primaryKey; // next object store key (id field)
    let value = cursor.value; // next object store object (book object)
    let key = cursor.key; // next index key (price)
    console.log(key, value);
    cursor.continue();
  } else {
    console.log("No more books");
  }
};
```

Promise wrapper

Adding `onsuccess/onerror` to every request is quite a cumbersome task. Sometimes we can make our life easier by using event delegation, e.g. set handlers on the whole transactions, but `async/await` is much more convenient.

Let's use a thin promise wrapper <https://github.com/jakearchibald/idb> further in this chapter. It creates a global `idb` object with `promisified` IndexedDB methods.

Then, instead of `onsuccess/onerror` we can write like this:

```
let db = await idb.openDb('store', 1, db => {
  if (db.oldVersion == 0) {
    // perform the initialization
    db.createObjectStore('books', {keyPath: 'id'});
  }
});

let transaction = db.transaction('books', 'readwrite');
let books = transaction.objectStore('books');

try {
  await books.add(...);
  await books.add(...);

  await transaction.complete;

  console.log('jsbook saved');
} catch(err) {
  console.log('error', err.message);
}
```

So we have all the sweet “plain async code” and “try...catch” stuff.

Error handling

If we don't catch an error, then it falls through, till the closest outer `try..catch`.

An uncaught error becomes an “unhandled promise rejection” event on `window` object.

We can handle such errors like this:

```
window.addEventListener('unhandledrejection', event => {
  let request = event.target; // IndexedDB native request object
  let error = event.reason; // Unhandled error object, same as request.error
  ...report about the error...
});
```

“Inactive transaction” pitfall

As we already know, a transaction auto-commits as soon as the browser is done with the current code and microtasks. So if we put a *macrotask* like `fetch` in the middle of a transaction, then the transaction won't wait for it to finish. It just auto-commits. So the next request in it would fail.

For a promise wrapper and `async/await` the situation is the same.

Here's an example of `fetch` in the middle of the transaction:

```
let transaction = db.transaction("inventory", "readwrite");
let inventory = transaction.objectStore("inventory");

await inventory.add({ id: 'js', price: 10, created: new Date() });

await fetch(...); // (*)

await inventory.add({ id: 'js', price: 10, created: new Date() }); // Error
```

The next `inventory.add` after `fetch (*)` fails with an “inactive transaction” error, because the transaction is already committed and closed at that time.

The workaround is same as when working with native IndexedDB: either make a new transaction or just split things apart.

1. Prepare the data and fetch all that's needed first.
2. Then save in the database.

Getting native objects

Internally, the wrapper performs a native IndexedDB request, adding `onerror/onsuccess` to it, and returns a promise that rejects/resolves with the result.

That works fine most of the time. The examples are at the lib page <https://github.com/jakearchibald/idb> ↗ .

In few rare cases, when we need the original `request` object, we can access it as `promise.request` property of the promise:

```
let promise = books.add(book); // get a promise (don't await for its result)

let request = promise.request; // native request object
let transaction = request.transaction; // native transaction object

// ...do some native IndexedDB voodoo...

let result = await promise; // if still needed
```

Summary

IndexedDB can be thought of as a “localStorage on steroids”. It’s a simple key-value database, powerful enough for offline apps, yet simple to use.

The best manual is the specification, [the current one](#) ↗ is 2.0, but few methods from [3.0](#) ↗ (it’s not much different) are partially supported.

The basic usage can be described with a few phrases:

1. Get a promise wrapper like [idb](#) ↗ .
2. Open a database: `idb.openDb(name, version, onupgradeneeded)`
 - Create object storages and indexes in `onupgradeneeded` handler or perform version update if needed.
3. For requests:
 - Create transaction `db.transaction('books')` (readwrite if needed).
 - Get the object store `transaction.objectStore('books')`.
4. Then, to search by a key, call methods on the object store directly.
 - To search by an object field, create an index.
5. If the data does not fit in memory, use a cursor.

Here’s a small demo app:

<https://plnkr.co/edit/QdxxpvCQ02wYrpxdGr3Sm?p=preview> ↗

Animation

CSS and JavaScript animations.

Bezier curve

Bezier curves are used in computer graphics to draw shapes, for CSS animation and in many other places.

They are a very simple thing, worth to study once and then feel comfortable in the world of vector graphics and advanced animations.

Control points

A [bezier curve](#) is defined by control points.

There may be 2, 3, 4 or more.

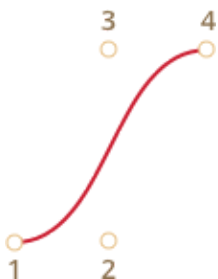
For instance, two points curve:



Three points curve:



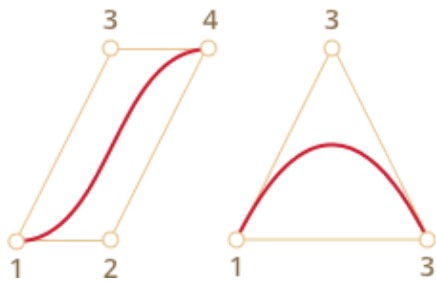
Four points curve:



If you look closely at these curves, you can immediately notice:

1. **Points are not always on curve.** That's perfectly normal, later we'll see how the curve is built.
2. **The curve order equals the number of points minus one.** For two points we have a linear curve (that's a straight line), for three points – quadratic curve (parabolic), for four points – cubic curve.

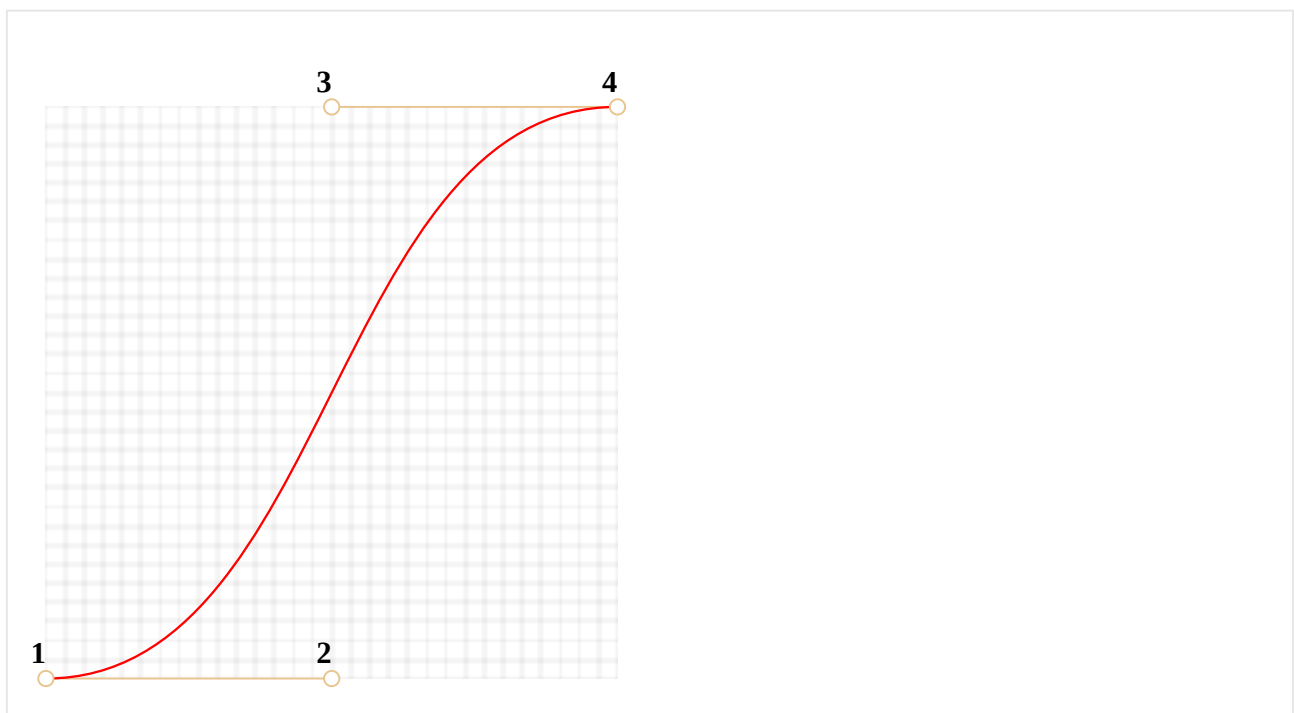
3. A curve is always inside the **convex hull** [↗](#) of control points:



Because of that last property, in computer graphics it's possible to optimize intersection tests. If convex hulls do not intersect, then curves do not either. So checking for the convex hulls intersection first can give a very fast “no intersection” result. Checking the intersection of convex hulls is much easier, because they are rectangles, triangles and so on (see the picture above), much simpler figures than the curve.

The main value of Bezier curves for drawing – by moving the points the curve is changing *in intuitively obvious way*.

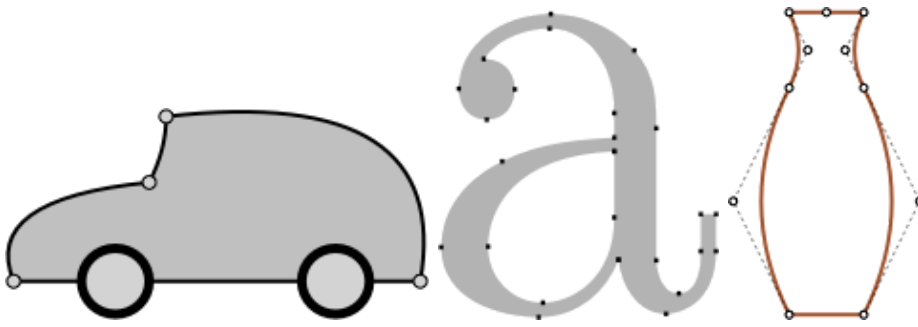
Try to move control points using a mouse in the example below:



As you can notice, the curve stretches along the tangential lines 1 → 2 and 3 → 4.

After some practice it becomes obvious how to place points to get the needed curve. And by connecting several curves we can get practically anything.

Here are some examples:



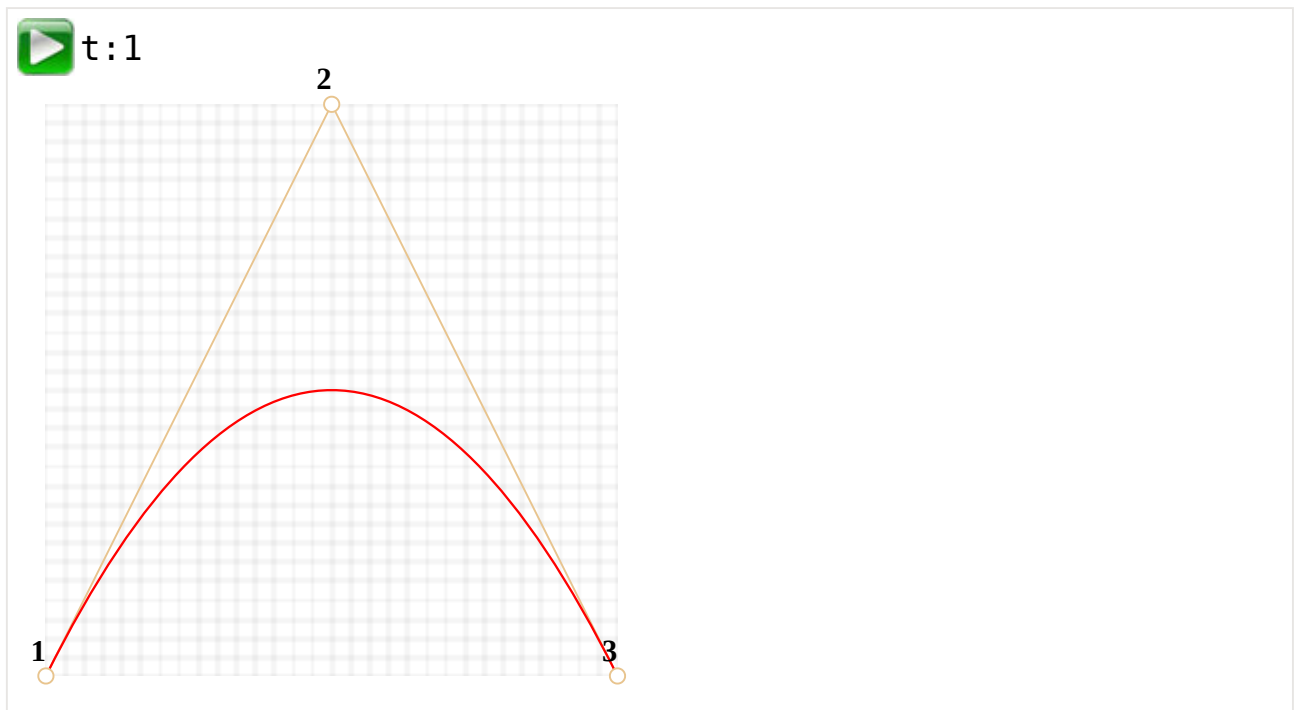
De Casteljau's algorithm

There's a mathematical formula for Bezier curves, but let's cover it a bit later, because [De Casteljau's algorithm](#) it is identical to the mathematical definition and visually shows how it is constructed.

First let's see the 3-points example.

Here's the demo, and the explanation follow.

Control points (1,2 and 3) can be moved by the mouse. Press the "play" button to run it.



De Casteljau's algorithm of building the 3-point bezier curve:

1. Draw control points. In the demo above they are labeled: 1, 2, 3.
2. Build segments between control points $1 \rightarrow 2 \rightarrow 3$. In the demo above they are brown.
3. The parameter t moves from 0 to 1. In the example above the step 0.05 is used: the loop goes over 0, 0.05, 0.1, 0.15, ... 0.95, 1.

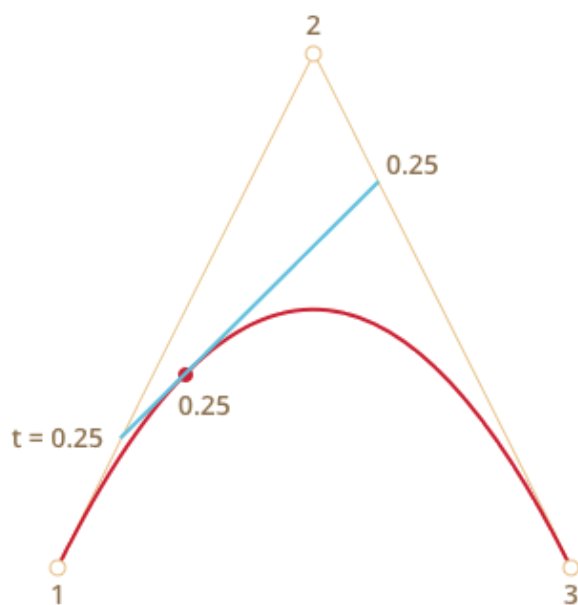
For each of these values of t :

- On each brown segment we take a point located on the distance proportional to t from its beginning. As there are two segments, we have two points.

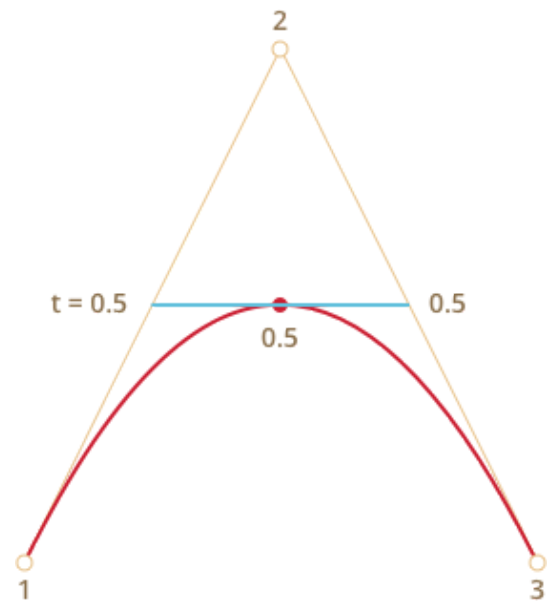
For instance, for $t=0$ – both points will be at the beginning of segments, and for $t=0.25$ – on the 25% of segment length from the beginning, for $t=0.5$ – 50%(the middle), for $t=1$ – in the end of segments.

- Connect the points. On the picture below the connecting segment is painted blue.

For $t=0.25$



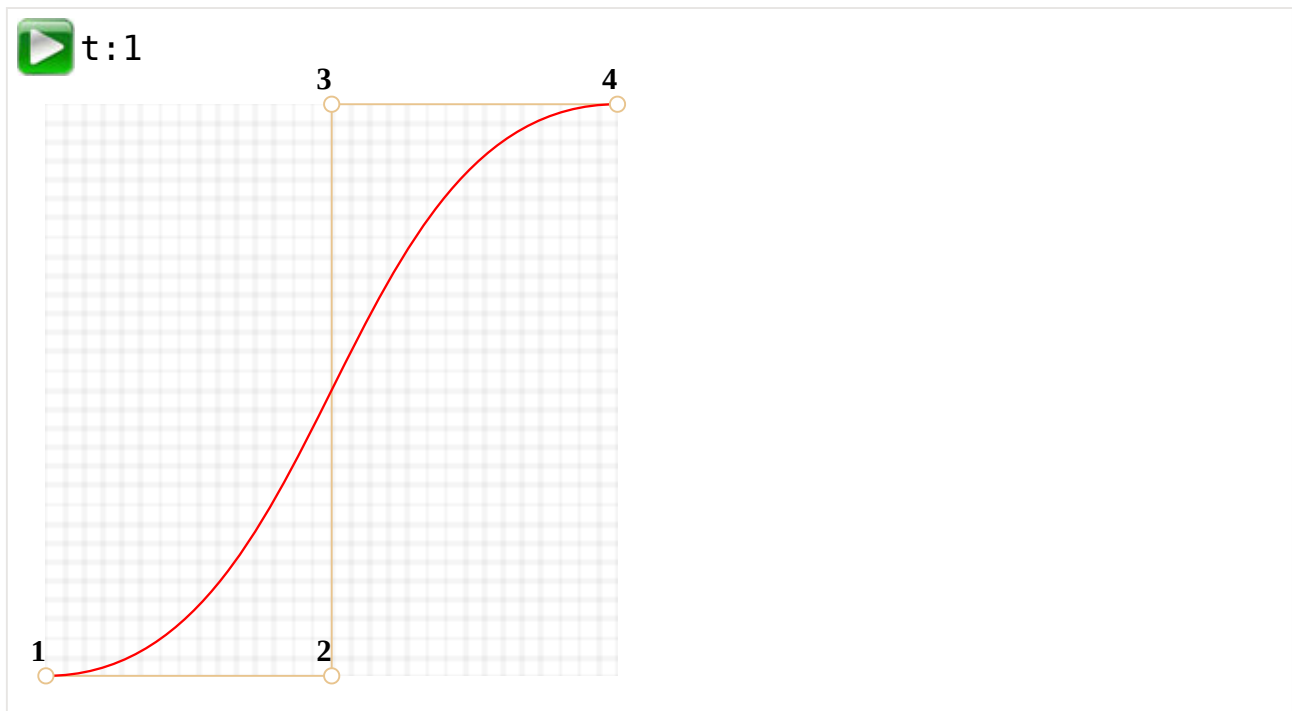
For $t=0.5$



- Now in the blue segment take a point on the distance proportional to the same value of t . That is, for $t=0.25$ (the left picture) we have a point at the end of the left quarter of the segment, and for $t=0.5$ (the right picture) – in the middle of the segment. On pictures above that point is red.
- As t runs from 0 to 1, every value of t adds a point to the curve. The set of such points forms the Bezier curve. It's red and parabolic on the pictures above.

That was a process for 3 points. But the same is for 4 points.

The demo for 4 points (points can be moved by a mouse):



The algorithm for 4 points:

- Connect control points by segments: $1 \rightarrow 2$, $2 \rightarrow 3$, $3 \rightarrow 4$. There will be 3 brown segments.
- For each t in the interval from 0 to 1:
 - We take points on these segments on the distance proportional to t from the beginning. These points are connected, so that we have two green segments.
 - On these segments we take points proportional to t . We get one blue segment.
 - On the blue segment we take a point proportional to t . On the example above it's red.
- These points together form the curve.

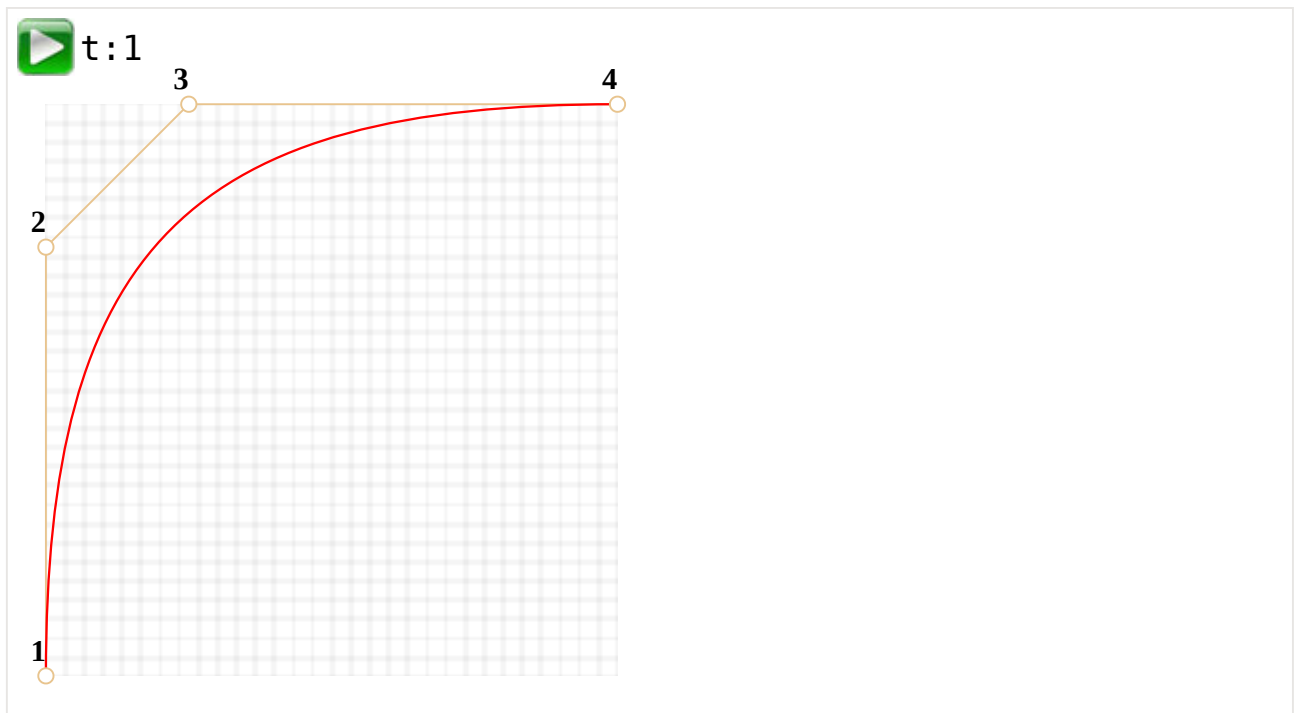
The algorithm is recursive and can be generalized for any number of control points.

Given N of control points:

1. We connect them to get initially $N-1$ segments.
2. Then for each t from 0 to 1, we take a point on each segment on the distance proportional to t and connect them. There will be $N-2$ segments.
3. Repeat step 2 until there is only one point.

These points make the curve.

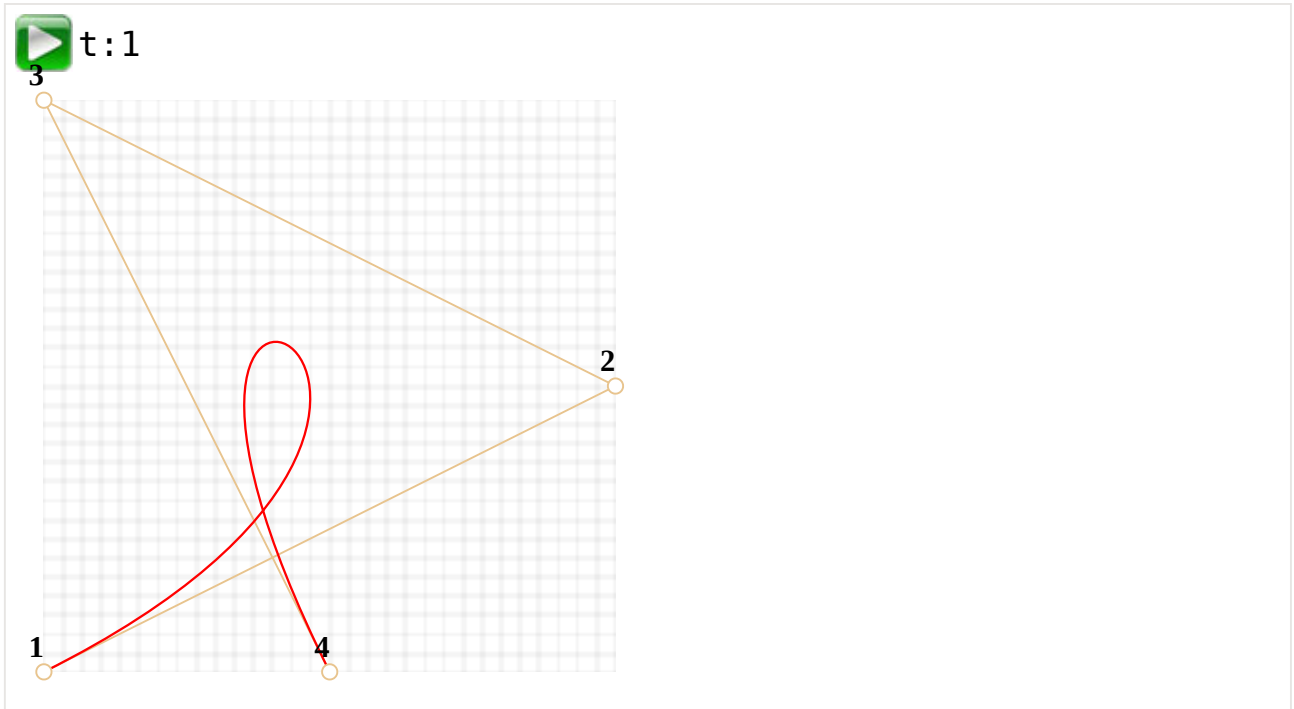
A curve that looks like $y=1/t$:



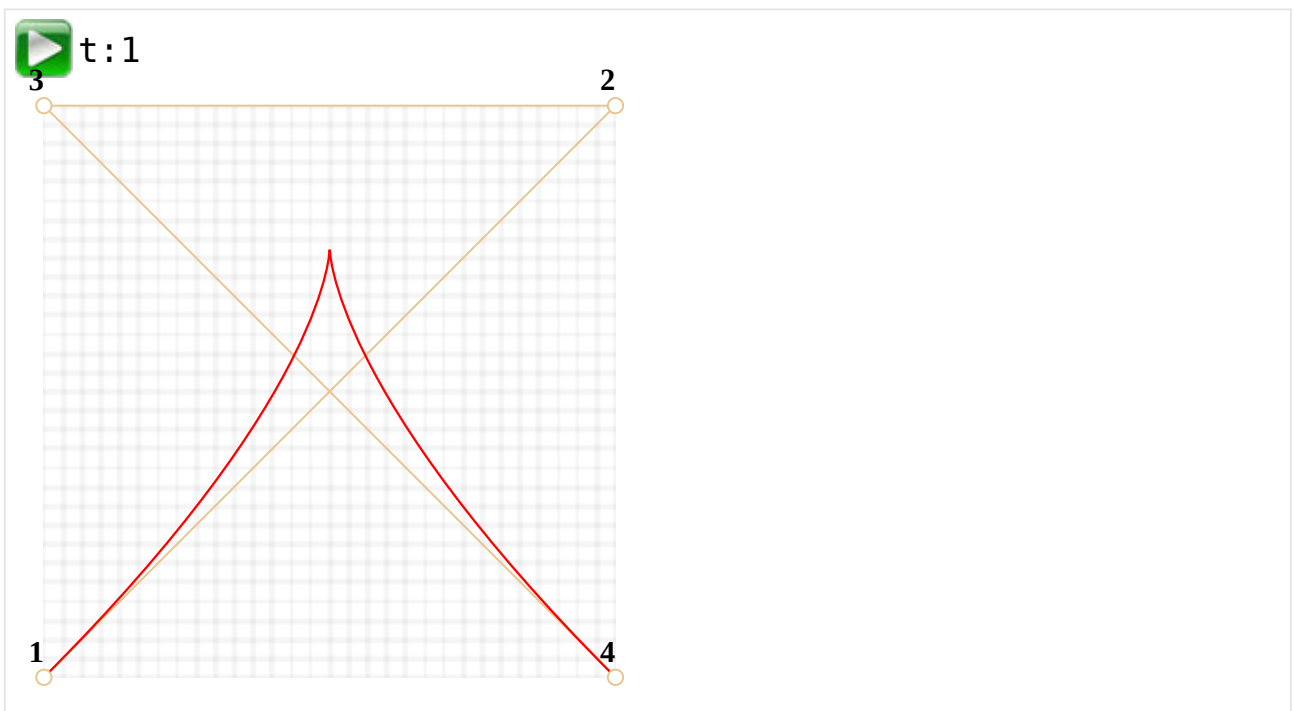
Zig-zag control points also work fine:



Making a loop is possible:



A non-smooth Bezier curve (yeah, that's possible too):



As the algorithm is recursive, we can build Bezier curves of any order, that is: using 5, 6 or more control points. But in practice many points are less useful. Usually we take 2-3 points, and for complex lines glue several curves together. That's simpler to develop and calculate.

i How to draw a curve *through* given points?

To specify a Bezier curve, control points are used. As we can see, they are not on the curve, except the first and the last ones.

Sometimes we have another task: to draw a curve *through several points*, so that all of them are on a single smooth curve. That task is called [interpolation](#) ↗ , and here we don't cover it.

There are mathematical formulas for such curves, for instance [Lagrange polynomial](#) ↗ . In computer graphics [spline interpolation](#) ↗ is often used to build smooth curves that connect many points.

Maths

A Bezier curve can be described using a mathematical formula.

As we saw – there's actually no need to know it, most people just draw the curve by moving points with a mouse. But if you're into maths – here it is.

Given the coordinates of control points P_i : the first control point has coordinates $P_1 = (x_1, y_1)$, the second: $P_2 = (x_2, y_2)$, and so on, the curve coordinates are described by the equation that depends on the parameter t from the segment $[0, 1]$.

- The formula for a 2-points curve:

$$P = (1-t)P_1 + tP_2$$

- For 3 control points:

$$P = (1-t)^2P_1 + 2(1-t)tP_2 + t^2P_3$$

- For 4 control points:

$$P = (1-t)^3P_1 + 3(1-t)^2tP_2 + 3(1-t)t^2P_3 + t^3P_4$$

These are vector equations. In other words, we can put x and y instead of P to get corresponding coordinates.

For instance, the 3-point curve is formed by points (x, y) calculated as:

- $x = (1-t)^2x_1 + 2(1-t)tx_2 + t^2x_3$
- $y = (1-t)^2y_1 + 2(1-t)ty_2 + t^2y_3$

Instead of $x_1, y_1, x_2, y_2, x_3, y_3$ we should put coordinates of 3 control points, and then as t moves from 0 to 1, for each value of t we'll have (x, y) of the curve.

For instance, if control points are $(0, 0)$, $(0.5, 1)$ and $(1, 0)$, the equations become:

- $x = (1-t)^2 * 0 + 2(1-t)t * 0.5 + t^2 * 1 = (1-t)t + t^2 = t$
- $y = (1-t)^2 * 0 + 2(1-t)t * 1 + t^2 * 0 = 2(1-t)t = -t^2 + 2t$

Now as t runs from 0 to 1, the set of values (x, y) for each t forms the curve for such control points.

Summary

Bezier curves are defined by their control points.

We saw two definitions of Bezier curves:

1. Using a drawing process: De Casteljau's algorithm.
2. Using a mathematical formulas.

Good properties of Bezier curves:

- We can draw smooth lines with a mouse by moving control points.
- Complex shapes can be made of several Bezier curves.

Usage:

- In computer graphics, modeling, vector graphic editors. Fonts are described by Bezier curves.
- In web development – for graphics on Canvas and in the SVG format. By the way, “live” examples above are written in SVG. They are actually a single SVG document that is given different points as parameters. You can open it in a separate window and see the source: [demo.svg](#).
- In CSS animation to describe the path and speed of animation.

CSS-animations

CSS animations allow to do simple animations without JavaScript at all.

JavaScript can be used to control CSS animation and make it even better with a little of code.

CSS transitions

The idea of CSS transitions is simple. We describe a property and how its changes should be animated. When the property changes, the browser paints the animation.

That is: all we need is to change the property. And the fluent transition is made by the browser.

For instance, the CSS below animates changes of `background-color` for 3 seconds:

```
.animated {  
  transition-property: background-color;  
  transition-duration: 3s;  
}
```

Now if an element has `.animated` class, any change of `background-color` is animated during 3 seconds.

Click the button below to animate the background:

```
<button id="color">Click me</button>  
  
<style>  
  #color {  
    transition-property: background-color;  
    transition-duration: 3s;  
  }  
</style>  
  
<script>  
  color.onclick = function() {  
    this.style.backgroundColor = 'red';  
  };  
</script>
```

Click me

There are 4 properties to describe CSS transitions:

- `transition-property`
- `transition-duration`
- `transition-timing-function`
- `transition-delay`

We'll cover them in a moment, for now let's note that the common `transition` property allows to declare them together in the order: `property duration`

timing-function delay , and also animate multiple properties at once.

For instance, this button animates both color and font-size :

```
<button id="growing">Click me</button>

<style>
#growing {
  transition: font-size 3s, color 2s;
}
</style>

<script>
growing.onclick = function() {
  this.style.fontSize = '36px';
  this.style.color = 'red';
};
</script>
```

Click me

Now let's cover animation properties one by one.

transition-property

In transition-property we write a list of property to animate, for instance: left, margin-left, height, color .

Not all properties can be animated, but [many of them](#) . The value all means “animate all properties”.

transition-duration

In transition-duration we can specify how long the animation should take. The time should be in [CSS time format](#) : in seconds s or milliseconds ms .

transition-delay

In transition-delay we can specify the delay *before* the animation. For instance, if transition-delay: 1s , then animation starts after 1 second after the change.

Negative values are also possible. Then the animation starts from the middle. For instance, if transition-duration is 2s , and the delay is -1s , then the

animation takes 1 second and starts from the half.

Here's the animation shifts numbers from 0 to 9 using CSS `translate` property:

<https://plnkr.co/edit/tRHA6fkSPUe9cjk35zPL?p=preview> ↗

The `transform` property is animated like this:

```
#stripe.animate {  
  transform: translate(-90%);  
  transition-property: transform;  
  transition-duration: 9s;  
}
```

In the example above JavaScript adds the class `.animate` to the element – and the animation starts:

```
stripe.classList.add('animate');
```

We can also start it “from the middle”, from the exact number, e.g. corresponding to the current second, using the negative `transition-delay`.

Here if you click the digit – it starts the animation from the current second:

<https://plnkr.co/edit/zpqja4CejOmTApUXPxE?p=preview> ↗

JavaScript does it by an extra line:

```
stripe.onclick = function() {  
  let sec = new Date().getSeconds() % 10;  
  // for instance, -3s here starts the animation from the 3rd second  
  stripe.style.transitionDelay = '-' + sec + 's';  
  stripe.classList.add('animate');  
};
```

transition-timing-function

Timing function describes how the animation process is distributed along the time. Will it start slowly and then go fast or vice versa.

That's the most complicated property from the first sight. But it becomes very simple if we devote a bit time to it.

That property accepts two kinds of values: a Bezier curve or steps. Let's start from the curve, as it's used more often.

Bezier curve

The timing function can be set as a [Bezier curve](#) with 4 control points that satisfies the conditions:

1. First control point: $(0, 0)$.
2. Last control point: $(1, 1)$.
3. For intermediate points values of x must be in the interval $0..1$, y can be anything.

The syntax for a Bezier curve in CSS: `cubic-bezier(x2, y2, x3, y3)`. Here we need to specify only 2nd and 3rd control points, because the 1st one is fixed to $(0, 0)$ and the 4th one is $(1, 1)$.

The timing function describes how fast the animation process goes in time.

- The x axis is the time: 0 – the starting moment, 1 – the last moment of `transition-duration`.
- The y axis specifies the completion of the process: 0 – the starting value of the property, 1 – the final value.

The simplest variant is when the animation goes uniformly, with the same linear speed. That can be specified by the curve `cubic-bezier(0, 0, 1, 1)`.

Here's how that curve looks:



...As we can see, it's just a straight line. As the time (x) passes, the completion (y) of the animation steadily goes from 0 to 1 .

The train in the example below goes from left to right with the permanent speed (click it):

<https://plnkr.co/edit/BKXxsW1mgxIZvhcpUcj4?p=preview>

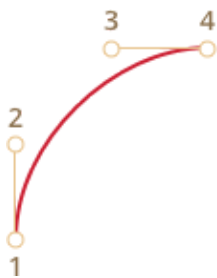
The CSS `transition` is based on that curve:

```
.train {  
  left: 0;  
  transition: left 5s cubic-bezier(0, 0, 1, 1);  
  /* JavaScript sets left to 450px */  
}
```

...And how can we show a train slowing down?

We can use another Bezier curve: `cubic-bezier(0.0, 0.5, 0.5, 1.0)`.

The graph:



As we can see, the process starts fast: the curve soars up high, and then slower and slower.

Here's the timing function in action (click the train):

<https://plnkr.co/edit/EBBtkTnl1l5096SHLcq8?p=preview>

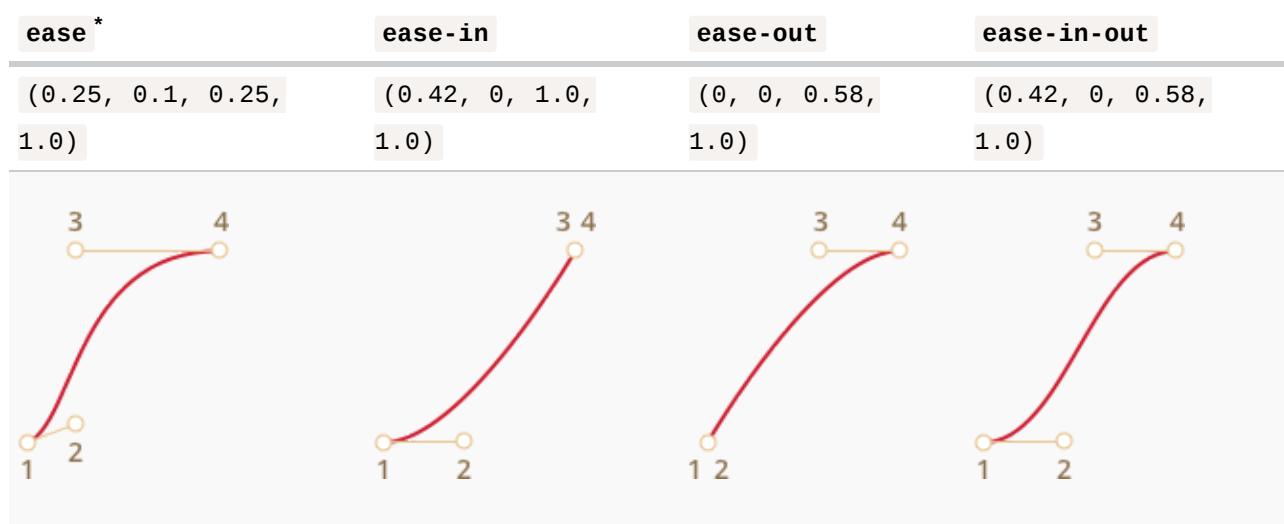
CSS:

```
.train {  
  left: 0;  
  transition: left 5s cubic-bezier(0, .5, .5, 1);  
  /* JavaScript sets left to 450px */  
}
```

There are several built-in curves: `linear`, `ease`, `ease-in`, `ease-out` and `ease-in-out`.

The `linear` is a shorthand for `cubic-bezier(0, 0, 1, 1)` – a straight line, we saw it already.

Other names are shorthands for the following `cubic-bezier`:



* – by default, if there's no timing function, `ease` is used.

So we could use `ease-out` for our slowing down train:

```
.train {
  left: 0;
  transition: left 5s ease-out;
  /* transition: left 5s cubic-bezier(0, .5, .5, 1); */
}
```

But it looks a bit differently.

A Bezier curve can make the animation “jump out” of its range.

The control points on the curve can have any `y` coordinates: even negative or huge. Then the Bezier curve would also jump very low or high, making the animation go beyond its normal range.

In the example below the animation code is:

```
.train {
  left: 100px;
  transition: left 5s cubic-bezier(.5, -1, .5, 2);
  /* JavaScript sets left to 400px */
}
```

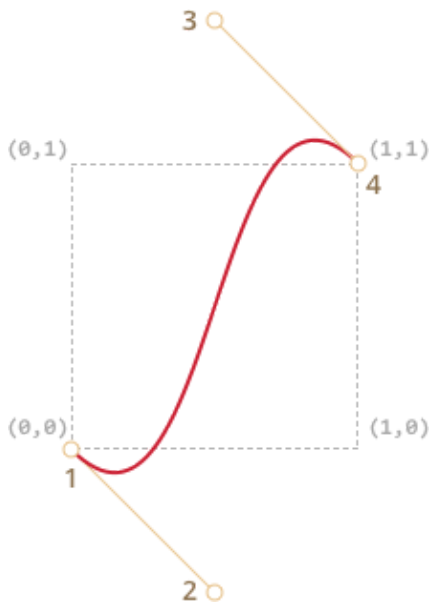
The property `left` should animate from `100px` to `400px`.

But if you click the train, you'll see that:

- First, the train goes *back*: `left` becomes less than `100px`.
- Then it goes forward, a little bit farther than `400px`.
- And then back again – to `400px`.

<https://plnkr.co/edit/TjXgcacdDDsFyYHb4Lnl?p=preview> ↗

Why it happens – pretty obvious if we look at the graph of the given Bezier curve:



We moved the `y` coordinate of the 2nd point below zero, and for the 3rd point we made put it over `1`, so the curve goes out of the “regular” quadrant. The `y` is out of the “standard” range `0..1`.

As we know, `y` measures “the completion of the animation process”. The value `y = 0` corresponds to the starting property value and `y = 1` – the ending value. So values `y < 0` move the property lower than the starting `left` and `y > 1` – over the final `left`.

That’s a “soft” variant for sure. If we put `y` values like `-99` and `99` then the train would jump out of the range much more.

But how to make the Bezier curve for a specific task? There are many tools. For instance, we can do it on the site <http://cubic-bezier.com/> ↗.

Steps

Timing function `steps(number of steps[, start/end])` allows to split animation into steps.

Let’s see that in an example with digits.

Here’s a list of digits, without any animations, just as a source:

<https://plnkr.co/edit/iyY2pj0vD8CcbFCuqnsI?p=preview> ↗

We’ll make the digits appear in a discrete way by making the part of the list outside of the red “window” invisible and shifting the list to the left with each step.

There will be 9 steps, a step-move for each digit:

```
#stripe.animate {
  transform: translate(-90%);
  transition: transform 9s steps(9, start);
}
```


In action:

<https://plnkr.co/edit/6VBxPjYIojjUL5vX8UvS?p=preview> ↗

The first argument of `steps(9, start)` is the number of steps. The transform will be split into 9 parts (10% each). The time interval is automatically divided into 9 parts as well, so `transition: 9s` gives us 9 seconds for the whole animation – 1 second per digit.

The second argument is one of two words: `start` or `end`.

The `start` means that in the beginning of animation we need to do make the first step immediately.

We can observe that during the animation: when we click on the digit it changes to `1` (the first step) immediately, and then changes in the beginning of the next second.

The process is progressing like this:

- `0s` – `-10%` (first change in the beginning of the 1st second, immediately)
- `1s` – `-20%`
- ...
- `8s` – `-80%`
- (the last second shows the final value).

The alternative value `end` would mean that the change should be applied not in the beginning, but at the end of each second.

So the process would go like this:

- `0s` – `0`
- `1s` – `-10%` (first change at the end of the 1st second)
- `2s` – `-20%`
- ...
- `9s` – `-90%`

Here's `step(9, end)` in action (note the pause between the first digit change):

<https://plnkr.co/edit/I3SoddMNBVDKxH2HeFak?p=preview> ↗

There are also shorthand values:

- `step-start` – is the same as `steps(1, start)`. That is, the animation starts immediately and takes 1 step. So it starts and finishes immediately, as if there were no animation.
- `step-end` – the same as `steps(1, end)`: make the animation in a single step at the end of `transition-duration`.

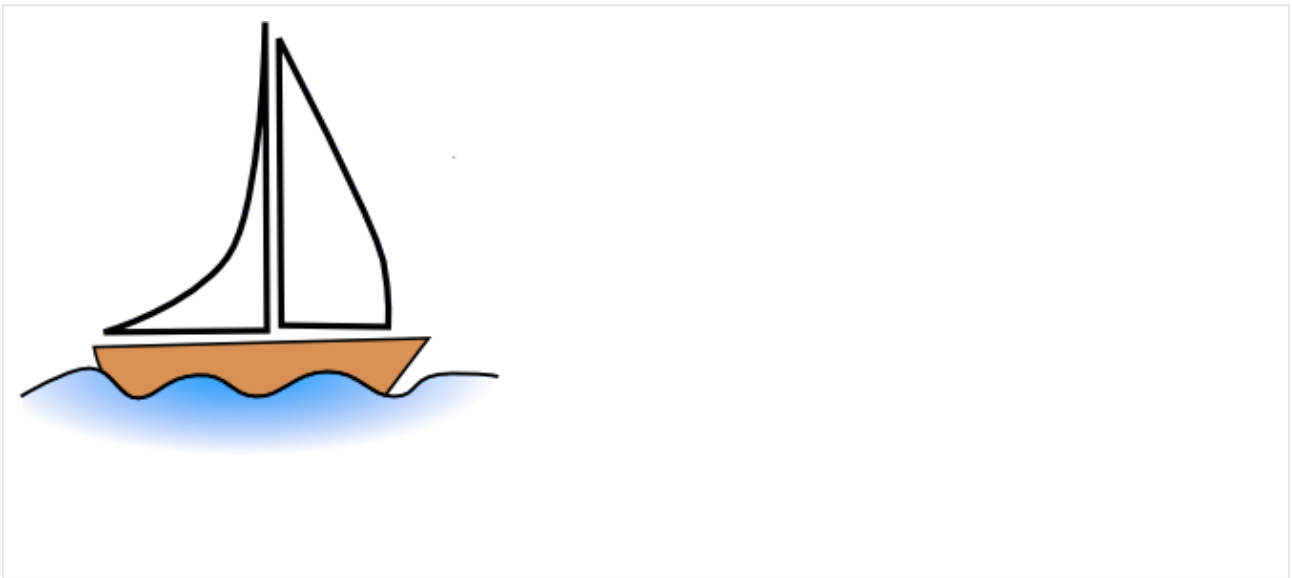
These values are rarely used, because that's not really animation, but rather a single-step change.

Event transitionend

When the CSS animation finishes the `transitionend` event triggers.

It is widely used to do an action after the animation is done. Also we can join animations.

For instance, the ship in the example below starts to swim there and back on click, each time farther and farther to the right:



The animation is initiated by the function `go` that re-runs each time when the transition finishes and flips the direction:

```
boat.onclick = function() {  
  //...  
  let times = 1;  
  
  function go() {  
    if (times % 2) {  
      // swim to the right  
      boat.classList.remove('back');  
      boat.style.marginLeft = 100 * times + 200 + 'px';  
    } else {  
      // swim to the left  
      boat.classList.add('back');  
      boat.style.marginLeft = 100 * times - 200 + 'px';  
    }  
  }  
  
  go();  
}
```

```
boat.addEventListener('transitionend', function() {
    times++;
    go();
});
};
```

The event object for `transitionend` has few specific properties:

`event.propertyName`

The property that has finished animating. Can be good if we animate multiple properties simultaneously.

`event.elapsedTime`

The time (in seconds) that the animation took, without `transition-delay`.

Keyframes

We can join multiple simple animations together using the `@keyframes` CSS rule.

It specifies the “name” of the animation and rules: what, when and where to animate. Then using the `animation` property we attach the animation to the element and specify additional parameters for it.

Here’s an example with explanations:

```
<div class="progress"></div>

<style>
    @keyframes go-left-right {          /* give it a name: "go-left-right" */
        from { left: 0px; }             /* animate from left: 0px */
        to { left: calc(100% - 50px); } /* animate to left: 100%-50px */
    }

    .progress {
        animation: go-left-right 3s infinite alternate;
        /* apply the animation "go-left-right" to the element
           duration 3 seconds
           number of times: infinite
           alternate direction every time
        */

        position: relative;
        border: 2px solid green;
        width: 50px;
        height: 20px;
        background: lime;
```

```
}  
</style>
```



There are many articles about `@keyframes` and a [detailed specification](#) .

Probably you won't need `@keyframes` often, unless everything is in the constant move on your sites.

Summary

CSS animations allow to smoothly (or not) animate changes of one or multiple CSS properties.

They are good for most animation tasks. We're also able to use JavaScript for animations, the next chapter is devoted to that.

Limitations of CSS animations compared to JavaScript animations:

Merits

- Simple things done simply.
- Fast and lightweight for CPU.

Demerits

- JavaScript animations are flexible. They can implement any animation logic, like an "explosion" of an element.
- Not just property changes. We can create new elements in JavaScript for purposes of animation.

The majority of animations can be implemented using CSS as described in this chapter. And `transitionend` event allows to run JavaScript after the animation, so it integrates fine with the code.

But in the next chapter we'll do some JavaScript animations to cover more complex cases.

✓ Tasks

Animate a plane (CSS)

importance: 5

Show the animation like on the picture below (click the plane):



- The picture grows on click from 40x24px to 400x240px (10 times larger).
- The animation takes 3 seconds.
- At the end output: “Done!”.
- During the animation process, there may be more clicks on the plane. They shouldn’t “break” anything.

[Open a sandbox for the task.](#) ↗

[To solution](#)

Animate the flying plane (CSS)

importance: 5

Modify the solution of the previous task [Animate a plane \(CSS\)](#) to make the plane grow more than it’s original size 400x240px (jump out), and then return to that size.

Here’s how it should look (click on the plane):



Take the solution of the previous task as the source.

[To solution](#)

Animated circle

importance: 5

Create a function `showCircle(cx, cy, radius)` that shows an animated growing circle.

- `cx, cy` are window-relative coordinates of the center of the circle,
- `radius` is the radius of the circle.

Click the button below to see how it should look like:

```
showCircle(150, 150, 100)
```

The source document has an example of a circle with right styles, so the task is precisely to do the animation right.

[Open a sandbox for the task.](#) ↗

[To solution](#)

JavaScript animations

JavaScript animations can handle things that CSS can't.

For instance, moving along a complex path, with a timing function different from Bezier curves, or an animation on a canvas.

Using setInterval

An animation can be implemented as a sequence of frames – usually small changes to HTML/CSS properties.

For instance, changing `style.left` from `0px` to `100px` moves the element. And if we increase it in `setInterval`, changing by `2px` with a tiny delay, like 50 times per second, then it looks smooth. That's the same principle as in the cinema: 24 frames per second is enough to make it look smooth.

The pseudo-code can look like this:

```
let timer = setInterval(function() {  
  if (animation complete) clearInterval(timer);  
  else increase style.left by 2px  
}, 20); // change by 2px every 20ms, about 50 frames per second
```

More complete example of the animation:

```
let start = Date.now(); // remember start time  
  
let timer = setInterval(function() {  
  // how much time passed from the start?  
  let timePassed = Date.now() - start;  
  
  if (timePassed >= 2000) {  
    clearInterval(timer); // finish the animation after 2 seconds  
    return;  
  }  
  
  // draw the animation at the moment timePassed  
  draw(timePassed);  
  
}, 20);
```

```
// as timePassed goes from 0 to 2000
// left gets values from 0px to 400px
function draw(timePassed) {
  train.style.left = timePassed / 5 + 'px';
}
```

Click for the demo:

<https://plnkr.co/edit/rah4Qvue9bwrmtS3ASKt?p=preview> ↗

Using requestAnimationFrame

Let's imagine we have several animations running simultaneously.

If we run them separately, then even though each one has `setInterval(..., 20)`, then the browser would have to repaint much more often than every `20ms`.

That's because they have different starting time, so "every 20ms" differs between different animations. The intervals are not aligned. So we'll have several independent runs within `20ms`.

In other words, this:

```
setInterval(function() {
  animate1();
  animate2();
  animate3();
}, 20)
```

...Is lighter than three independent calls:

```
setInterval(animate1, 20); // independent animations
setInterval(animate2, 20); // in different places of the script
setInterval(animate3, 20);
```

These several independent redraws should be grouped together, to make the redraw easier for the browser and hence load less CPU load and look smoother.

There's one more thing to keep in mind. Sometimes when CPU is overloaded, or there are other reasons to redraw less often (like when the browser tab is hidden), so we really shouldn't run it every `20ms`.

But how do we know about that in JavaScript? There's a specification [Animation timing](#) ↗ that provides the function `requestAnimationFrame`. It addresses all these issues and even more.

The syntax:


```
let requestId = requestAnimationFrame(callback)
```

That schedules the `callback` function to run in the closest time when the browser wants to do animation.

If we do changes in elements in `callback` then they will be grouped together with other `requestAnimationFrame` callbacks and with CSS animations. So there will be one geometry recalculation and repaint instead of many.

The returned value `requestId` can be used to cancel the call:

```
// cancel the scheduled execution of callback  
cancelAnimationFrame(requestId);
```

The `callback` gets one argument – the time passed from the beginning of the page load in microseconds. This time can also be obtained by calling [performance.now\(\)](#) .

Usually `callback` runs very soon, unless the CPU is overloaded or the laptop battery is almost discharged, or there's another reason.

The code below shows the time between first 10 runs for `requestAnimationFrame`. Usually it's 10-20ms:

```
<script>  
  let prev = performance.now();  
  let times = 0;  
  
  requestAnimationFrame(function measure(time) {  
    document.body.insertAdjacentHTML("beforeEnd", Math.floor(time - prev) + " ");  
    prev = time;  
  
    if (times++ < 10) requestAnimationFrame(measure);  
  })  
</script>
```

Structured animation

Now we can make a more universal animation function based on `requestAnimationFrame`:

```
function animate({timing, draw, duration}) {

  let start = performance.now();

  requestAnimationFrame(function animate(time) {
    // timeFraction goes from 0 to 1
    let timeFraction = (time - start) / duration;
    if (timeFraction > 1) timeFraction = 1;

    // calculate the current animation state
    let progress = timing(timeFraction)

    draw(progress); // draw it

    if (timeFraction < 1) {
      requestAnimationFrame(animate);
    }

  });
}
```

Function `animate` accepts 3 parameters that essentially describes the animation:

duration

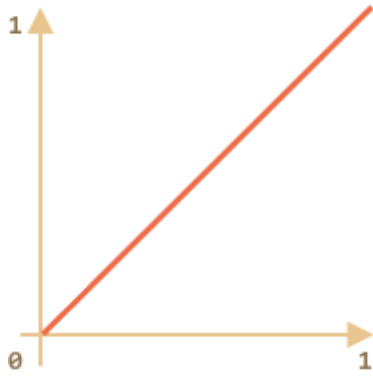
Total time of animation. Like, `1000` .

timing(timeFraction)

Timing function, like CSS-property `transition-timing-function` that gets the fraction of time that passed (`0` at start, `1` at the end) and returns the animation completion (like `y` on the Bezier curve).

For instance, a linear function means that the animation goes on uniformly with the same speed:

```
function linear(timeFraction) {
  return timeFraction;
}
```



It's graph: 0

1

That's just like `transition-timing-function: linear`. There are more interesting variants shown below.

`draw(progress)`

The function that takes the animation completion state and draws it. The value `progress=0` denotes the beginning animation state, and `progress=1` – the end state.

This is that function that actually draws out the animation.

It can move the element:

```
function draw(progress) {  
  train.style.left = progress + 'px';  
}
```

...Or do anything else, we can animate anything, in any way.

Let's animate the element `width` from `0` to `100%` using our function.

Click on the element for the demo:

<https://plnkr.co/edit/5l241DCQmzPNofVr2wfk?p=preview>

The code for it:

```
animate({  
  duration: 1000,  
  timing(timeFraction) {  
    return timeFraction;  
  },  
  draw(progress) {  
    elem.style.width = progress * 100 + '%';  
  }  
});
```

Unlike CSS animation, we can make any timing function and any drawing function here. The timing function is not limited by Bezier curves. And `draw` can go beyond properties, create new elements for like fireworks animation or something.

Timing functions

We saw the simplest, linear timing function above.

Let's see more of them. We'll try movement animations with different timing functions to see how they work.

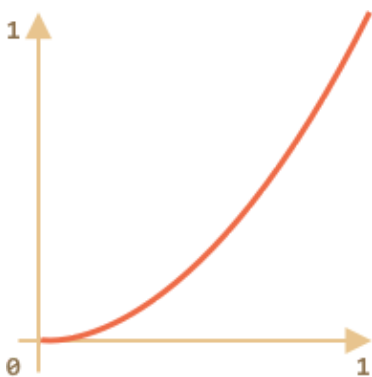
Power of n

If we want to speed up the animation, we can use `progress` in the power `n`.

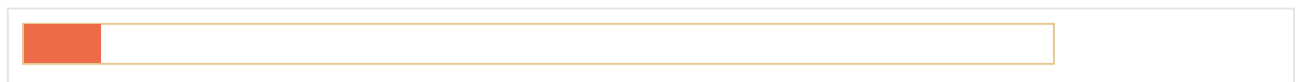
For instance, a parabolic curve:

```
function quad(timeFraction) {  
  return Math.pow(timeFraction, 2)  
}
```

The graph:

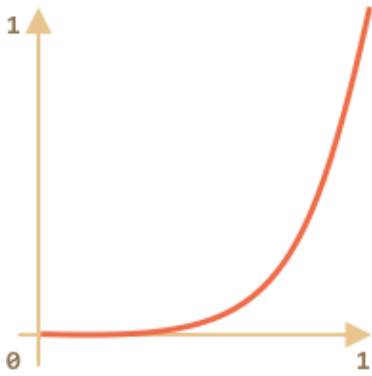


See in action (click to activate):

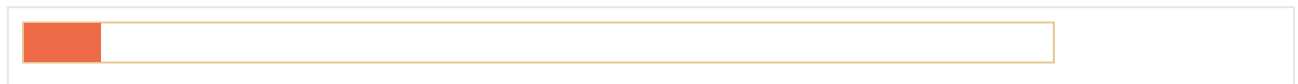


...Or the cubic curve or even greater `n`. Increasing the power makes it speed up faster.

Here's the graph for `progress` in the power `5`:



In action:

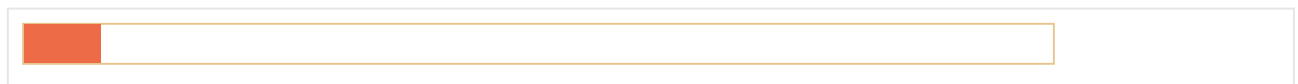
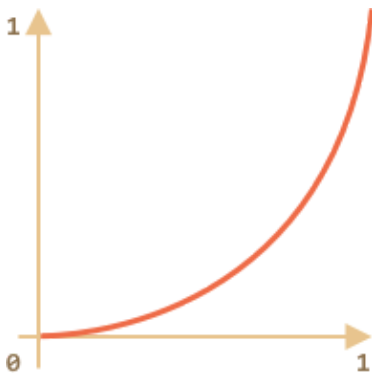


The arc

Function:

```
function circ(timeFraction) {
  return 1 - Math.sin(Math.acos(timeFraction));
}
```

The graph:



Back: bow shooting

This function does the “bow shooting”. First we “pull the bowstring”, and then “shoot”.

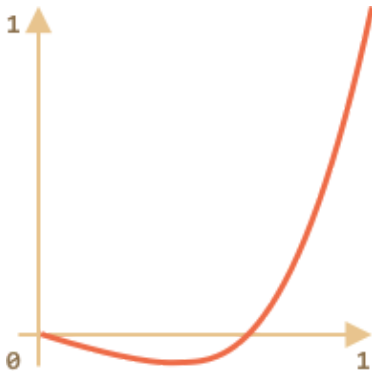
Unlike previous functions, it depends on an additional parameter `x`, the “elasticity coefficient”. The distance of “bowstring pulling” is defined by it.

The code:

```
function back(x, timeFraction) {
  return Math.pow(timeFraction, 2) * ((x + 1) * timeFraction - x)
```

```
}
```

The graph for $x = 1.5$:



For animation we use it with a specific value of x . Example for $x = 1.5$:



Bounce

Imagine we are dropping a ball. It falls down, then bounces back a few times and stops.

The `bounce` function does the same, but in the reverse order: “bouncing” starts immediately. It uses few special coefficients for that:

```
function bounce(timeFraction) {  
  for (let a = 0, b = 1, result; 1; a += b, b /= 2) {  
    if (timeFraction >= (7 - 4 * a) / 11) {  
      return -Math.pow((11 - 6 * a - 11 * timeFraction) / 4, 2) + Math.pow(b, 2)  
    }  
  }  
}
```

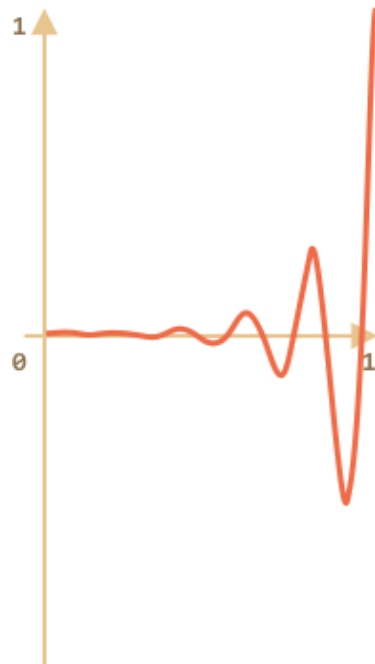
In action:



Elastic animation

One more “elastic” function that accepts an additional parameter x for the “initial range”.

```
function elastic(x, timeFraction) {  
  return Math.pow(2, 10 * (timeFraction - 1)) * Math.cos(20 * Math.PI * x / 3 * timeFraction)  
}
```



The graph for `x=1.5` :

In action for `x=1.5` :



Reversal: ease*

So we have a collection of timing functions. Their direct application is called “easeIn”.

Sometimes we need to show the animation in the reverse order. That’s done with the “easeOut” transform.

easeOut

In the “easeOut” mode the `timing` function is put into a wrapper

`timingEaseOut` :

```
timingEaseOut(timeFraction) = 1 - timing(1 - timeFraction)
```

In other words, we have a “transform” function `makeEaseOut` that takes a “regular” timing function and returns the wrapper around it:

```
// accepts a timing function, returns the transformed variant
function makeEaseOut(timing) {
  return function(timeFraction) {
    return 1 - timing(1 - timeFraction);
  }
}
```

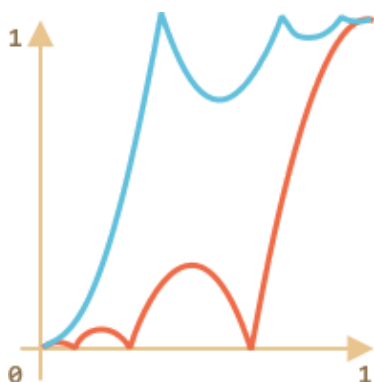
For instance, we can take the `bounce` function described above and apply it:

```
let bounceEaseOut = makeEaseOut(bounce);
```

Then the bounce will be not in the beginning, but at the end of the animation. Looks even better:

<https://plnkr.co/edit/opuSRjafyQ8Y41QXOoID?p=preview> ↗

Here we can see how the transform changes the behavior of the function:



If there's an animation effect in the beginning, like bouncing – it will be shown at the end.

In the graph above the **regular bounce** has the red color, and the **easeOut bounce** is blue.

- Regular bounce – the object bounces at the bottom, then at the end sharply jumps to the top.
- After `easeOut` – it first jumps to the top, then bounces there.

easeInOut

We also can show the effect both in the beginning and the end of the animation. The transform is called “easeInOut”.

Given the timing function, we calculate the animation state like this:

```
if (timeFraction <= 0.5) { // first half of the animation
  return timing(2 * timeFraction) / 2;
} else { // second half of the animation
  return (2 - timing(2 * (1 - timeFraction))) / 2;
}
```

The wrapper code:


```
function makeEaseInOut(timing) {
  return function(timeFraction) {
    if (timeFraction < .5)
      return timing(2 * timeFraction) / 2;
    else
      return (2 - timing(2 * (1 - timeFraction))) / 2;
  }
}

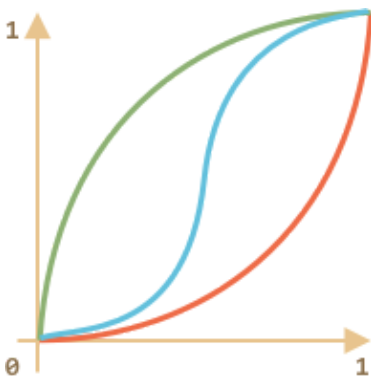
bounceEaseInOut = makeEaseInOut(bounce);
```

In action, `bounceEaseInOut` :

<https://plnkr.co/edit/F7NuLTRQbIC8EgZr8ltV?p=preview>

The “easeInOut” transform joins two graphs into one: `easeIn` (regular) for the first half of the animation and `easeOut` (reversed) – for the second part.

The effect is clearly seen if we compare the graphs of `easeIn` , `easeOut` and `easeInOut` of the `circ` timing function:



- Red is the regular variant of `circ` (`easeIn`).
- Green – `easeOut` .
- Blue – `easeInOut` .

As we can see, the graph of the first half of the animation is the scaled down `easeIn` , and the second half is the scaled down `easeOut` . As a result, the animation starts and finishes with the same effect.

More interesting “draw”

Instead of moving the element we can do something else. All we need is to write the proper `draw` .

Here’s the animated “bouncing” text typing:

<https://plnkr.co/edit/IX995Jbip9id4R2Vwer9?p=preview>

Summary

For animations that CSS can't handle well, or those that need tight control, JavaScript can help. JavaScript animations should be implemented via `requestAnimationFrame`. That built-in method allows to setup a callback function to run when the browser will be preparing a repaint. Usually that's very soon, but the exact time depends on the browser.

When a page is in the background, there are no repaints at all, so the callback won't run: the animation will be suspended and won't consume resources. That's great.

Here's the helper `animate` function to setup most animations:

```
function animate({timing, draw, duration}) {  
  
  let start = performance.now();  
  
  requestAnimationFrame(function animate(time) {  
    // timeFraction goes from 0 to 1  
    let timeFraction = (time - start) / duration;  
    if (timeFraction > 1) timeFraction = 1;  
  
    // calculate the current animation state  
    let progress = timing(timeFraction);  
  
    draw(progress); // draw it  
  
    if (timeFraction < 1) {  
      requestAnimationFrame(animate);  
    }  
  
  });  
}
```

Options:

- `duration` – the total animation time in ms.
- `timing` – the function to calculate animation progress. Gets a time fraction from 0 to 1, returns the animation progress, usually from 0 to 1.
- `draw` – the function to draw the animation.

Surely we could improve it, add more bells and whistles, but JavaScript animations are not applied on a daily basis. They are used to do something interesting and non-standard. So you'd want to add the features that you need when you need them.

JavaScript animations can use any timing function. We covered a lot of examples and transformations to make them even more versatile. Unlike CSS, we are not limited to Bezier curves here.

The same is about `draw` : we can animate anything, not just CSS properties.

✓ Tasks

Animate the bouncing ball

importance: 5

Make a bouncing ball. Click to see how it should look:



[Open a sandbox for the task.](#) [↗](#)

[To solution](#)

Animate the ball bouncing to the right

importance: 5

Make the ball bounce to the right. Like this:



Write the animation code. The distance to the left is `100px` .

Take the solution of the previous task [Animate the bouncing ball](#) as the source.

[To solution](#)

Web components

Web components is a set of standards to make self-contained components: custom HTML-elements with their own properties and methods, encapsulated DOM and styles.

From the orbital height

This section describes a set of modern standards for “web components”.

As of now, these standards are under development. Some features are well-supported and integrated into the modern HTML/DOM standard, while others are yet in draft stage. You can try examples in any browser, Google Chrome is probably the most up to date with these features. Guess, that’s because Google fellows are behind many of the related specifications.

What’s common between...

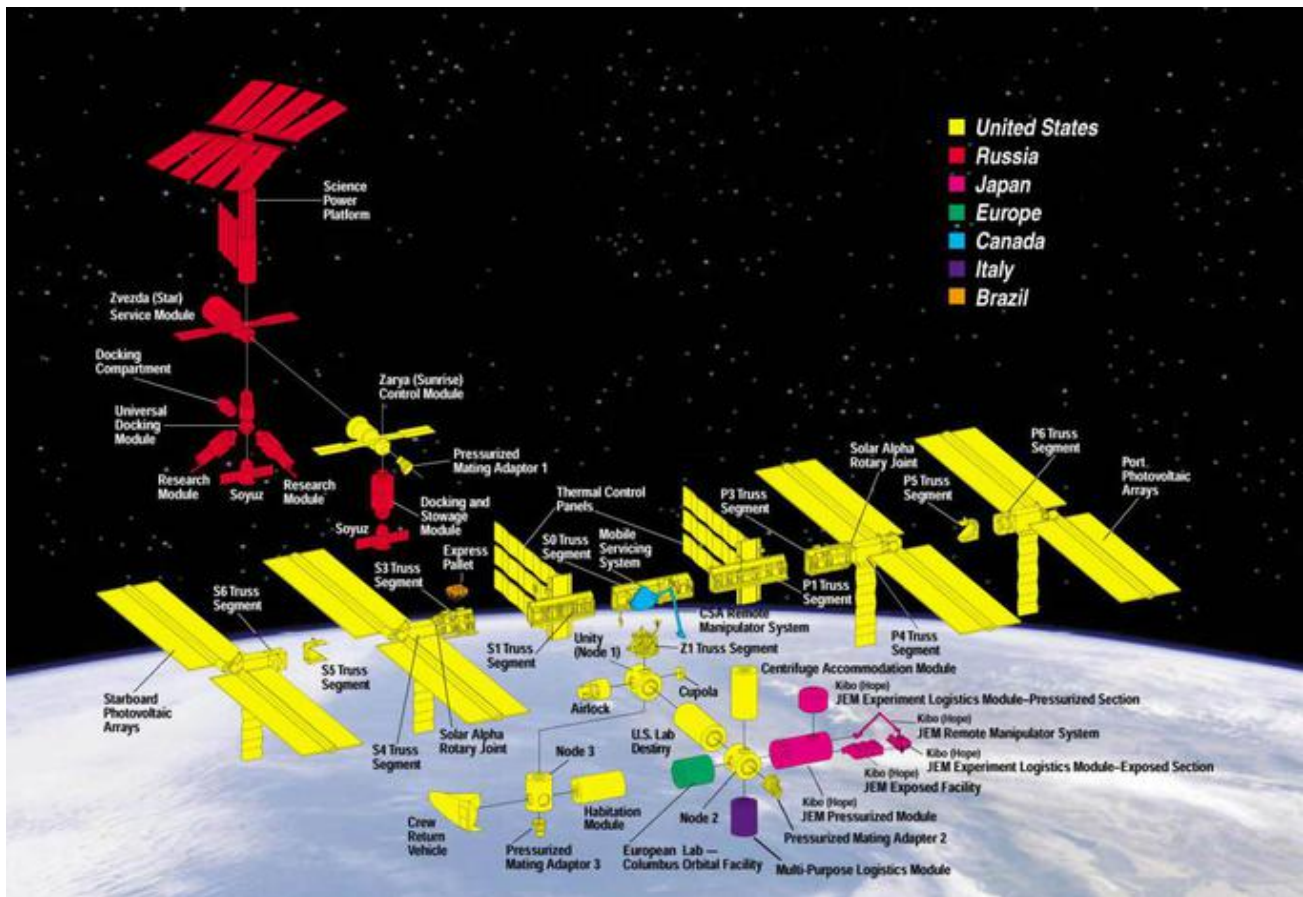
The whole component idea is nothing new. It’s used in many frameworks and elsewhere.

Before we move to implementation details, take a look at this great achievement of humanity:



That's the International Space Station (ISS).

And this is how it's made inside (approximately):



The International Space Station:

- Consists of many components.
- Each component, in its turn, has many smaller details inside.
- The components are very complex, much more complicated than most websites.
- Components are developed internationally, by teams from different countries, speaking different languages.

...And this thing flies, keeps humans alive in space!

How such complex devices are created?

Which principles we could borrow to make our development same-level reliable and scalable? Or, at least, close to it.

Component architecture

The well known rule for developing complex software is: don't make complex software.

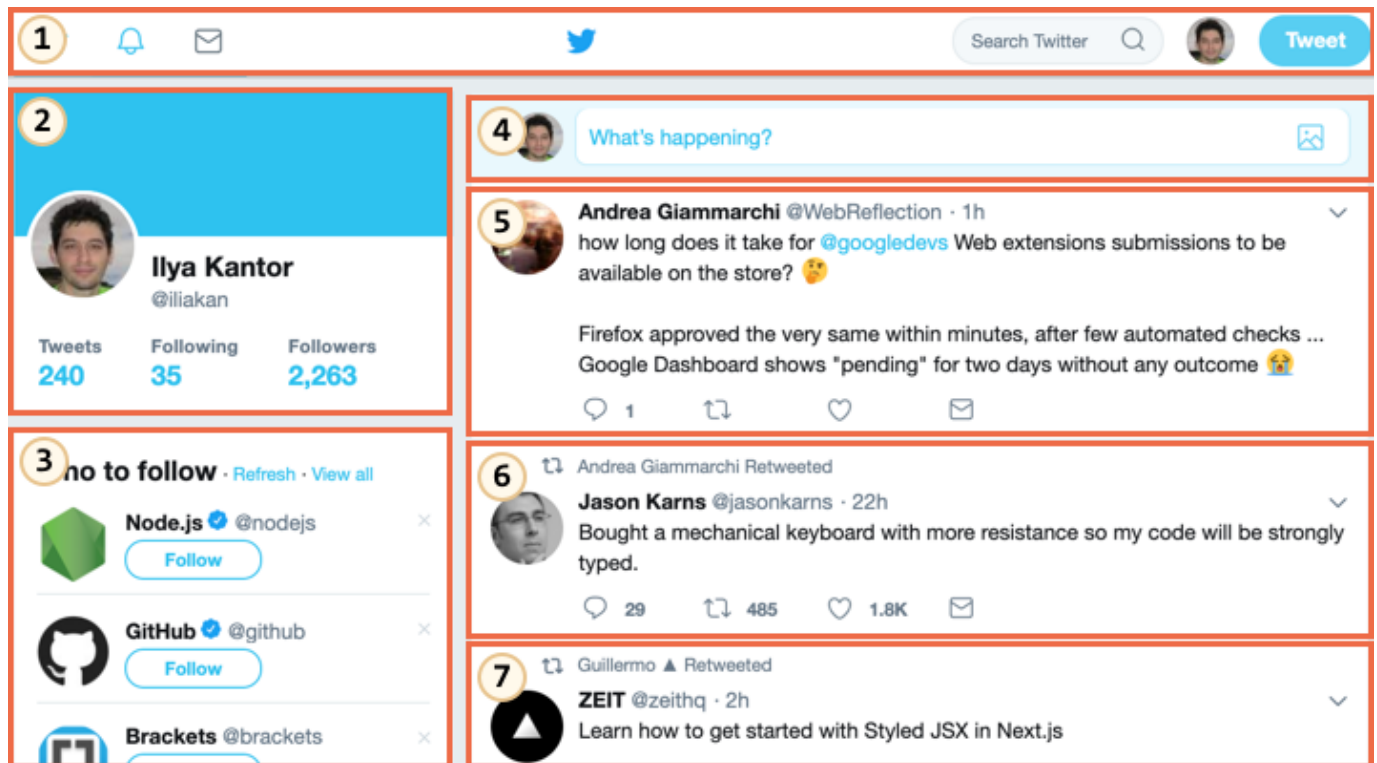
If something becomes complex – split it into simpler parts and connect in the most obvious way.

A good architect is the one who can make the complex simple.

We can split user interface into visual components: each of them has own place on the page, can “do” a well-described task, and is separate from the others.

Let’s take a look at a website, for example Twitter.

It naturally splits into components:



1. Top navigation.
2. User info.
3. Follow suggestions.
4. Submit form.
5. (and also 6, 7) – messages.

Components may have subcomponents, e.g. messages may be parts of a higher-level “message list” component. A clickable user picture itself may be a component, and so on.

How do we decide, what is a component? That comes from intuition, experience and common sense. Usually it’s a separate visual entity that we can describe in terms of what it does and how it interacts with the page. In the case above, the page has blocks, each of them plays its own role, it’s logical to make these components.

A component has:

- its own JavaScript class.
- DOM structure, managed solely by its class, outside code doesn’t access it (“encapsulation” principle).
- CSS styles, applied to the component.

- API: events, class methods etc, to interact with other components.

Once again, the whole “component” thing is nothing special.

There exist many frameworks and development methodologies to build them, each one with its own bells and whistles. Usually, special CSS classes and conventions are used to provide “component feel” – CSS scoping and DOM encapsulation.

“Web components” provide built-in browser capabilities for that, so we don’t have to emulate them any more.

- [Custom elements](#) – to define custom HTML elements.
- [Shadow DOM](#) – to create an internal DOM for the component, hidden from the others.
- [CSS Scoping](#) – to declare styles that only apply inside the Shadow DOM of the component.
- [Event retargeting](#) and other minor stuff to make custom components better fit the development.

In the next chapter we’ll go into details of “Custom Elements” – the fundamental and well-supported feature of web components, good on its own.

Custom elements

We can create custom HTML elements, described by our class, with its own methods and properties, events and so on.

Once an custom element is defined, we can use it on par with built-in HTML elements.

That’s great, as HTML dictionary is rich, but not infinite. There are no `<easy-tabs>`, `<sliding-carousel>`, `<beautiful-upload>` ... Just think of any other tag we might need.

We can define them with a special class, and then use as if they were always a part of HTML.

There are two kinds of custom elements:

1. **Autonomous custom elements** – “all-new” elements, extending the abstract `HTMLElement` class.
2. **Customized built-in elements** – extending built-in elements, like customized `HTMLButtonElement` etc.

First we’ll create autonomous elements, and then customized built-in ones.

To create a custom element, we need to tell the browser several details about it: how to show it, what to do when the element is added or removed to page, etc.

That's done by making a class with special methods. That's easy, as there are only few methods, and all of them are optional.

Here's a sketch with the full list:

```
class MyElement extends HTMLElement {
  constructor() {
    super();
    // element created
  }

  connectedCallback() {
    // browser calls it when the element is added to the document
    // (can be called many times if an element is repeatedly added/removed)
  }

  disconnectedCallback() {
    // browser calls it when the element is removed from the document
    // (can be called many times if an element is repeatedly added/removed)
  }

  static get observedAttributes() {
    return [/* array of attribute names to monitor for changes */];
  }

  attributeChangedCallback(name, oldValue, newValue) {
    // called when one of attributes listed above is modified
  }

  adoptedCallback() {
    // called when the element is moved to a new document
    // (happens in document.adoptNode, very rarely used)
  }

  // there can be other element methods and properties
}
```

After that, we need to register the element:

```
// let the browser know that <my-element> is served by our new class
customElements.define("my-element", MyElement);
```

Now for any HTML elements with tag `<my-element>`, an instance of `MyElement` is created, and the aforementioned methods are called. We also can `document.createElement('my-element')` in JavaScript.

i Custom element name must contain a hyphen -

Custom element name must have a hyphen -, e.g. `my-element` and `super-button` are valid names, but `myelement` is not.

That's to ensure that there are no name conflicts between built-in and custom HTML elements.

Example: “time-formatted”

For example, there already exists `<time>` element in HTML, for date/time. But it doesn't do any formatting by itself.

Let's create `<time-formatted>` element that displays the time in a nice, language-aware format:

```
<script>
class TimeFormatted extends HTMLElement { // (1)

  connectedCallback() {
    let date = new Date(this.getAttribute('datetime') || Date.now());

    this.innerHTML = new Intl.DateTimeFormat("default", {
      year: this.getAttribute('year') || undefined,
      month: this.getAttribute('month') || undefined,
      day: this.getAttribute('day') || undefined,
      hour: this.getAttribute('hour') || undefined,
      minute: this.getAttribute('minute') || undefined,
      second: this.getAttribute('second') || undefined,
      timeZoneName: this.getAttribute('time-zone-name') || undefined,
    }).format(date);
  }
}

customElements.define("time-formatted", TimeFormatted); // (2)
</script>

<!-- (3) -->
<time-formatted datetime="2019-12-01"
  year="numeric" month="long" day="numeric"
  hour="numeric" minute="numeric" second="numeric"
  time-zone-name="short"
></time-formatted>
```

December 1, 2019, 3:00:00 AM GMT+3

1. The class has only one method `connectedCallback()` – the browser calls it when `<time-formatted>` element is added to page (or when HTML parser detects it), and it uses the built-in [Intl.DateTimeFormat](#) data formatter, well-supported across the browsers, to show a nicely formatted time.
2. We need to register our new element by `customElements.define(tag, class)`.
3. And then we can use it everywhere.

Custom elements upgrade

If the browser encounters any `<time-formatted>` elements before `customElements.define`, that's not an error. But the element is yet unknown, just like any non-standard tag.

Such “undefined” elements can be styled with CSS selector `:not(:defined)`.

When `customElement.define` is called, they are “upgraded”: a new instance of `TimeFormatted` is created for each, and `connectedCallback` is called. They become `:defined`.

To get the information about custom elements, there are methods:

- `customElements.get(name)` – returns the class for a custom element with the given `name`,
- `customElements.whenDefined(name)` – returns a promise that resolves (without value) when a custom element with the given `name` becomes defined.

i Rendering in `connectedCallback`, not in `constructor`

In the example above, element content is rendered (created) in `connectedCallback`.

Why not in the `constructor`?

The reason is simple: when `constructor` is called, it's yet too early. The element instance is created, but not populated yet. The browser did not yet process/assign attributes at this stage: calls to `getAttribute` would return `null`. So we can't really render there.

Besides, if you think about it, that's better performance-wise – to delay the work until it's really needed.

The `connectedCallback` triggers when the element is added to the document. Not just appended to another element as a child, but actually becomes a part of the page. So we can build detached DOM, create elements and prepare them for later use. They will only be actually rendered when they make it into the page.

Observing attributes

In the current implementation of `<time-formatted>`, after the element is rendered, further attribute changes don't have any effect. That's strange for an HTML element. Usually, when we change an attribute, like `a.href`, we expect the change to be immediately visible. So let's fix this.

We can observe attributes by providing their list in `observedAttributes()` static getter. For such attributes, `attributeChangedCallback` is called when they are modified. It doesn't trigger for an attribute for performance reasons.

Here's a new `<time-formatted>`, that auto-updates when attributes change:

```
<script>
class TimeFormatted extends HTMLElement {

  render() { // (1)
    let date = new Date(this.getAttribute('datetime') || Date.now());

    this.innerHTML = new Intl.DateTimeFormat("default", {
      year: this.getAttribute('year') || undefined,
      month: this.getAttribute('month') || undefined,
      day: this.getAttribute('day') || undefined,
      hour: this.getAttribute('hour') || undefined,
      minute: this.getAttribute('minute') || undefined,
      second: this.getAttribute('second') || undefined,
      timeZoneName: this.getAttribute('time-zone-name') || undefined,
```

```

    }).format(date);
  }

  connectedCallback() { // (2)
    if (!this.rendered) {
      this.render();
      this.rendered = true;
    }
  }

  static get observedAttributes() { // (3)
    return ['datetime', 'year', 'month', 'day', 'hour', 'minute', 'second', 'time-zo
  }

  attributeChangedCallback(name, oldValue, newValue) { // (4)
    this.render();
  }
}

customElements.define("time-formatted", TimeFormatted);
</script>

<time-formatted id="elem" hour="numeric" minute="numeric" second="numeric"></time-f
</script>
setInterval(() => elem.setAttribute('datetime', new Date()), 1000); // (5)
</script>

```

7:14:48 PM

1. The rendering logic is moved to `render()` helper method.
2. We call it once when the element is inserted into page.
3. For a change of an attribute, listed in `observedAttributes()`, `attributeChangedCallback` triggers.
4. ...and re-renders the element.
5. At the end, we can easily make a live timer.

Rendering order

When HTML parser builds the DOM, elements are processed one after another, parents before children. E.g. if we have `<outer><inner></inner></outer>`, then `<outer>` element is created and connected to DOM first, and then `<inner>`.

That leads to important consequences for custom elements.

For example, if a custom element tries to access `innerHTML` in `connectedCallback`, it gets nothing:

```

<script>
customElements.define('user-info', class extends HTMLElement {

  connectedCallback() {
    alert(this.innerHTML); // empty (*)
  }

});
</script>

<user-info>John</user-info>

```

If you run it, the `alert` is empty.

That's exactly because there are no children on that stage, the DOM is unfinished. HTML parser connected the custom element `<user-info>`, and will now proceed to its children, but just didn't yet.

If we'd like to pass information to custom element, we can use attributes. They are available immediately.

Or, if we really need the children, we can defer access to them with zero-delay `setTimeout`.

This works:

```

<script>
customElements.define('user-info', class extends HTMLElement {

  connectedCallback() {
    setTimeout(() => alert(this.innerHTML)); // John (*)
  }

});
</script>

<user-info>John</user-info>

```

Now the `alert` in line `(*)` shows "John", as we run it asynchronously, after the HTML parsing is complete. We can process children if needed and finish the initialization.

On the other hand, this solution is also not perfect. If nested custom elements also use `setTimeout` to initialize themselves, then they queue up: the outer `setTimeout` triggers first, and then the inner one.

So the outer element finishes the initialization before the inner one.

Let's demonstrate that on example:

```
<script>
customElements.define('user-info', class extends HTMLElement {
  connectedCallback() {
    alert(`${this.id} connected.`);
    setTimeout(() => alert(`${this.id} initialized.`));
  }
});
</script>
```

```
<user-info id="outer">
  <user-info id="inner"></user-info>
</user-info>
```

Output order:

1. outer connected.
2. inner connected.
3. outer initialized.
4. inner initialized.

We can clearly see that the outer element does not wait for the inner one.

There's no built-in callback that triggers after nested elements are ready. But we can implement such thing on our own. For instance, inner elements can dispatch events like `initialized`, and outer ones can listen and react on them.

Customized built-in elements

New elements that we create, such as `<time-formatted>`, don't have any associated semantics. They are unknown to search engines, and accessibility devices can't handle them.

But such things can be important. E.g, a search engine would be interested to know that we actually show a time. And if we're making a special kind of button, why not reuse the existing `<button>` functionality?

We can extend and customize built-in elements by inheriting from their classes.

For example, buttons are instances of `HTMLButtonElement`, let's build upon it.

1. Extend `HTMLButtonElement` with our class:

```
class HelloButton extends HTMLButtonElement { /* custom element methods */ }
```

2. Provide an third argument to `customElements.define`, that specifies the tag:

```
customElements.define('hello-button', HelloButton, {extends: 'button'});
```

There exist different tags that share the same class, that's why it's needed.

3. At the end, to use our custom element, insert a regular `<button>` tag, but add `is="hello-button"` to it:

```
<button is="hello-button">...</button>
```

Here's a full example:

```
<script>
// The button that says "hello" on click
class HelloButton extends HTMLElement {
  constructor() {
    super();
    this.addEventListener('click', () => alert("Hello!"));
  }
}

customElements.define('hello-button', HelloButton, {extends: 'button'});
</script>

<button is="hello-button">Click me</button>

<button is="hello-button" disabled>Disabled</button>
```



Our new button extends the built-in one. So it keeps the same styles and standard features like `disabled` attribute.

References

- HTML Living Standard: <https://html.spec.whatwg.org/#custom-elements> .
- Compatibility: <https://caniuse.com/#feat=custom-elements> .

Summary

Custom elements can be of two types:

1. “Autonomous” – new tags, extending `HTMLElement` .

Definition scheme:


```
class MyElement extends HTMLElement {
  constructor() { super(); /* ... */ }
  connectedCallback() { /* ... */ }
  disconnectedCallback() { /* ... */ }
  static get observedAttributes() { return []; /* ... */ }
  attributeChangedCallback(name, oldValue, newValue) { /* ... */ }
  adoptedCallback() { /* ... */ }
}
customElements.define('my-element', MyElement);
/* <my-element> */
```

2. “Customized built-in elements” – extensions of existing elements.

Requires one more `.define` argument, and `is="..."` in HTML:

```
class MyButton extends HTMLButtonElement { /*...*/ }
customElements.define('my-button', MyElement, {extends: 'button'});
/* <button is="my-button"> */
```

Custom elements are well-supported among browsers. Edge is a bit behind, but there’s a polyfill <https://github.com/webcomponents/webcomponentsjs> .

✓ Tasks

Live timer element

We already have `<time-formatted>` element to show a nicely formatted time.

Create `<live-timer>` element to show the current time:

1. It should use `<time-formatted>` internally, not duplicate its functionality.
2. Ticks (updates) every second.
3. For every tick, a custom event named `tick` should be generated, with the current date in `event.detail` (see chapter [Dispatching custom events](#)).

Usage:

```
<live-timer id="elem"></live-timer>

<script>
  elem.addEventListener('tick', event => console.log(event.detail));
</script>
```

Demo:

7:14:48 PM

[Open a sandbox for the task.](#) ↗

[To solution](#)

Shadow DOM

Shadow DOM serves for encapsulation. It allows a component to have its very own “shadow” DOM tree, that can’t be accidentally accessed from the main document, may have local style rules, and more.

Built-in shadow DOM

Did you ever think how complex browser controls are created and styled?

Such as `<input type="range">`:



The browser uses DOM/CSS internally to draw them. That DOM structure is normally hidden from us, but we can see it in developer tools. E.g. in Chrome, we need to enable in Dev Tools “Show user agent shadow DOM” option.

Then `<input type="range">` looks like this:

```
▼<input type="range"> == $0
  ▼#shadow-root (user-agent)
    ▼<div>
      ▼<div pseudo="-webkit-slider-runnable-track" id="track">
        <div id="thumb"></div>
      </div>
    </div>
  </input>
```

What you see under `#shadow-root` is called “shadow DOM”.

We can’t get built-in shadow DOM elements by regular JavaScript calls or selectors. These are not regular children, but a powerful encapsulation technique.


In the example above, we can see a useful attribute `pseudo`. It’s non-standard, exists for historical reasons. We can use it style subelements with CSS, like this:

```
<style>
/* make the slider track red */
input::-webkit-slider-runnable-track {
  background: red;
```

```
}  
</style>  
  
<input type="range">
```



Once again, `pseudo` is a non-standard attribute. Chronologically, browsers first started to experiment with internal DOM structures to implement controls, and then, after time, shadow DOM was standardized to allow us, developers, to do the similar thing.

Further on, we'll use the modern shadow DOM standard, covered by [DOM spec](#)  other related specifications.

Shadow tree

A DOM element can have two types of DOM subtrees:

1. Light tree – a regular DOM subtree, made of HTML children. All subtrees that we've seen in previous chapters were “light”.
2. Shadow tree – a hidden DOM subtree, not reflected in HTML, hidden from prying eyes.

If an element has both, then the browser renders only the shadow tree. But we can setup a kind of composition between shadow and light trees as well. We'll see the details later in the chapter [Shadow DOM slots, composition](#).

Shadow tree can be used in Custom Elements to hide component internals and apply component-local styles.

For example, this `<show-hello>` element hides its internal DOM in shadow tree:

```
<script>  
customElements.define('show-hello', class extends HTMLElement {  
  connectedCallback() {  
    const shadow = this.attachShadow({mode: 'open'});  
    shadow.innerHTML = `<p>  
      Hello, ${this.getAttribute('name')}  
    </p>`;  
  }  
});  
</script>  
  
<show-hello name="John"></show-hello>
```

Hello, John

That's how the resulting DOM looks in Chrome dev tools, all the content is under “#shadow-root”:

```
▼ <show-hello name="John"> == $0
  ▼ #shadow-root (open)
    <p>Hello, John!</p>
  </show-hello>
```

First, the call to `elem.attachShadow({mode: ...})` creates a shadow tree.

There are two limitations:

1. We can create only one shadow root per element.
2. The `elem` must be either a custom element, or one of: “article”, “aside”, “blockquote”, “body”, “div”, “footer”, “h1...h6”, “header”, “main”, “nav”, “p”, “section”, or “span”. Other elements, like ``, can't host shadow tree.

The `mode` option sets the encapsulation level. It must have any of two values:

- “open” – the shadow root is available as `elem.shadowRoot`.

Any code is able to access the shadow tree of `elem`.

- “closed” – `elem.shadowRoot` is always `null`.

We can only access the shadow DOM by the reference returned by `attachShadow` (and probably hidden inside a class). Browser-native shadow trees, such as `<input type="range">`, are closed. There's no way to access them.

The [shadow root](#), returned by `attachShadow`, is like an element: we can use `innerHTML` or DOM methods, such as `append`, to populate it.

The element with a shadow root is called a “shadow tree host”, and is available as the shadow root `host` property:

```
// assuming {mode: "open"}, otherwise elem.shadowRoot is null
alert(elem.shadowRoot.host === elem); // true
```

Encapsulation

Shadow DOM is strongly delimited from the main document:

1. Shadow DOM elements are not visible to `querySelector` from the light DOM.
In particular, Shadow DOM elements may have ids that conflict with those in the light DOM. They must be unique only within the shadow tree.
2. Shadow DOM has own stylesheets. Style rules from the outer DOM don't get applied.

For example:

```
<style>
  /* document style won't apply to the shadow tree inside #elem (1) */
  p { color: red; }
</style>

<div id="elem"></div>

<script>
  elem.attachShadow({mode: 'open'});
  // shadow tree has its own style (2)
  elem.shadowRoot.innerHTML = `
    <style> p { font-weight: bold; } </style>
    <p>Hello, John!</p>
  `;

  // <p> is only visible from queries inside the shadow tree (3)
  alert(document.querySelectorAll('p').length); // 0
  alert(elem.shadowRoot.querySelectorAll('p').length); // 1
</script>
```

1. The style from the document does not affect the shadow tree.
2. ...But the style from the inside works.
3. To get elements in shadow tree, we must query from inside the tree.

References

- DOM: <https://dom.spec.whatwg.org/#shadow-trees> ↗
- Compatibility: <https://caniuse.com/#feat=shadowdomv1> ↗
- Shadow DOM is mentioned in many other specifications, e.g. [DOM Parsing](#) ↗ specifies that shadow root has `innerHTML`.

Summary

Shadow DOM is a way to create a component-local DOM.

1. `shadowRoot = elem.attachShadow({mode: open|closed})` – creates shadow DOM for `elem`. If `mode="open"`, then it's accessible as

`elem.shadowRoot` property.

2. We can populate `shadowRoot` using `innerHTML` or other DOM methods.

Shadow DOM elements:

- Have their own ids space,
- Invisible to JavaScript selectors from the main document, such as `querySelector`,
- Use styles only from the shadow tree, not from the main document.

Shadow DOM, if exists, is rendered by the browser instead of so-called “light DOM” (regular children). In the chapter [Shadow DOM slots, composition](#) we’ll see how to compose them.

Template element

A built-in `<template>` element serves as a storage for HTML markup templates. The browser ignores its contents, only checks for syntax validity, but we can access and use it in JavaScript, to create other elements.

In theory, we could create any invisible element somewhere in HTML for HTML markup storage purposes. What’s special about `<template>`?

First, its content can be any valid HTML, even if it normally requires a proper enclosing tag.

For example, we can put there a table row `<tr>`:

```
<template>
  <tr>
    <td>Contents</td>
  </tr>
</template>
```

Usually, if we try to put `<tr>` inside, say, a `<div>`, the browser detects the invalid DOM structure and “fixes” it, adds `<table>` around. That’s not what we want. On the other hand, `<template>` keeps exactly what we place there.

We can put styles and scripts into `<template>` as well:

```
<template>
  <style>
    p { font-weight: bold; }
  </style>
  <script>
```

```
    alert("Hello");  
  </script>  
</template>
```

The browser considers `<template>` content “out of the document”: styles are not applied, scripts are not executed, `<video autoplay>` is not run, etc.

The content becomes live (styles apply, scripts run etc) when we insert it into the document.

Inserting template

The template content is available in its `content` property as a [DocumentFragment](#) – a special type of DOM node.

We can treat it as any other DOM node, except one special property: when we insert it somewhere, its children are inserted instead.

For example:

```
<template id="tpl1">  
  <script>  
    alert("Hello");  
  </script>  
  <div class="message">Hello, world!</div>  
</template>  
  
<script>  
  let elem = document.createElement('div');  
  
  // Clone the template content to reuse it multiple times  
  elem.append(tmp1.content.cloneNode(true));  
  
  document.body.append(elem);  
  // Now the script from <template> runs  
</script>
```

Let's rewrite a Shadow DOM example from the previous chapter using `<template>`:

```
<template id="tpl1">  
  <style> p { font-weight: bold; } </style>  
  <p id="message"></p>  
</template>  
  
<div id="elem">Click me</div>
```

```

<script>
  elem.onclick = function() {
    elem.attachShadow({mode: 'open'});

    elem.shadowRoot.append(tmp1.content.cloneNode(true)); // (*)

    elem.shadowRoot.getElementById('message').innerHTML = "Hello from the shadows!"
  };
</script>

```

Click me

In the line `(*)` when we clone and insert `tmp1.content`, as it's `DocumentFragment`, its children (`<style>`, `<p>`) are inserted instead.

They form the shadow DOM:

```

<div id="elem">
  #shadow-root
    <style> p { font-weight: bold; } </style>
    <p id="message"></p>
</div>

```

Summary

To summarize:

- `<template>` content can be any syntactically correct HTML.
- `<template>` content is considered “out of the document”, so it doesn't affect anything.
- We can access `template.content` from JavaScript, clone it to reuse in a new component.

The `<template>` tag is quite unique, because:

- The browser checks HTML syntax inside it (as opposed to using a template string inside a script).
- ...But still allows to use any top-level HTML tags, even those that don't make sense without proper wrappers (e.g. `<tr>`).
- The content becomes interactive: scripts run, `<video autoplay>` plays etc, when inserted into the document.

The `<template>` element does not feature any iteration mechanisms, data binding or variable substitutions, but we can implement those on top of it.

Shadow DOM slots, composition

Many types of components, such as tabs, menus, image galleries, and so on, need the content to render.

Just like built-in browser `<select>` expects `<option>` items, our `<custom-tabs>` may expect the actual tab content to be passed. And a `<custom-menu>` may expect menu items.

The code that makes use of `<custom-menu>` can look like this:

```
<custom-menu>
  <title>Candy menu</title>
  <item>Lollipop</item>
  <item>Fruit Toast</item>
  <item>Cup Cake</item>
</custom-menu>
```

...Then our component should render it properly, as a nice menu with given title and items, handle menu events, etc.

How to implement it?

We could try to analyze the element content and dynamically copy-rearrange DOM nodes. That's possible, but if we're moving elements to shadow DOM, then CSS styles from the document do not apply in there, so the visual styling may be lost. Also that requires some coding.

Luckily, we don't have to. Shadow DOM supports `<slot>` elements, that are automatically filled by the content from light DOM.

Named slots

Let's see how slots work on a simple example.

Here, `<user-card>` shadow DOM provides two slots, filled from light DOM:

```
<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <div>Name:
        <slot name="username"></slot>
      </div>
      <div>Birthday:
        <slot name="birthday"></slot>
      </div>
```

```

    }
  });
</script>

<user-card>
  <span slot="username">John Smith</span>
  <span slot="birthday">01.01.2001</span>
</user-card>

```

Name: John Smith
 Birthday: 01.01.2001

In the shadow DOM, `<slot name="X">` defines an “insertion point”, a place where elements with `slot="X"` are rendered.

Then the browser performs “composition”: it takes elements from the light DOM and renders them in corresponding slots of the shadow DOM. At the end, we have exactly what we want – a generic component that can be filled with data.

Here’s the DOM structure after the script, not taking composition into account:

```

<user-card>
  #shadow-root
    <div>Name:
      <slot name="username"></slot>
    </div>
    <div>Birthday:
      <slot name="birthday"></slot>
    </div>
    <span slot="username">John Smith</span>
    <span slot="birthday">01.01.2001</span>
</user-card>

```

There’s nothing odd here. We created the shadow DOM, so here it is. Now the element has both light and shadow DOM.

For rendering purposes, for each `<slot name="...">` in shadow DOM, the browser looks for `slot="..."` with the same name in the light DOM. These elements are rendered inside the slots:

```

<user-card>
  #shadow-root
    <div>Name:
      <slot name="username"></slot>
    </div>
    <div>Birthday:
      <slot name="birthday"></slot>
    </div>
    <span slot="username">John Smith</span>
    <span slot="birthday">01.01.2001</span>
</user-card>

```



The result is called “flattened” DOM:

```

<user-card>
  #shadow-root
    <div>Name:
      <slot name="username">
        <!-- slotted element is inserted into the slot as a whole -->
        <span slot="username">John Smith</span>
      </slot>
    </div>
    <div>Birthday:
      <slot name="birthday">
        <span slot="birthday">01.01.2001</span>
      </slot>
    </div>
</user-card>

```

...But the “flattened” DOM is only created for rendering and event-handling purposes. That’s how things are shown. The nodes are actually not moved around!

That can be easily checked if we run `querySelector`: nodes are still at their places.

```

// light DOM <span> nodes are still at the same place, under `<user-card>`
alert( document.querySelector('user-card span').length ); // 2

```

It may look bizarre, but for shadow DOM with slots we have one more “DOM level”, the “flattened” DOM – result of slot insertion. The browser renders it and uses for style inheritance, event propagation. But JavaScript still sees the document “as is”, before flattening.

⚠️ Only top-level children may have slot="..." attribute

The `slot="..."` attribute is only valid for direct children of the shadow host (in our example, `<user-card>` element). For nested elements it's ignored.

For example, the second `` here is ignored (as it's not a top-level child of `<user-card>`):

```
<user-card>
  <span slot="username">John Smith</span>
  <div>
    <!-- bad slot, not top-level: -->
    <span slot="birthday">01.01.2001</span>
  </div>
</user-card>
```

In practice, there's no sense in slotting a deeply nested element, so this limitation just ensures the correct DOM structure.

Slot fallback content

If we put something inside a `<slot>`, it becomes the fallback content. The browser shows it if there's no corresponding filler in light DOM.

For example, in this piece of shadow DOM, `Anonymous` renders if there's no `slot="username"` in light DOM.

```
<div>Name:
  <slot name="username">Anonymous</slot>
</div>
```

Default slot

The first `<slot>` in shadow DOM that doesn't have a name is a "default" slot. It gets all nodes from the light DOM that aren't slotted elsewhere.

For example, let's add the default slot to our `<user-card>` that collects any unslotted information about the user:

```
<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
```

```

    <div>Name:
      <slot name="username"></slot>
    </div>
    <div>Birthday:
      <slot name="birthday"></slot>
    </div>
    <fieldset>
      <legend>Other information</legend>
      <slot></slot>
    </fieldset>
  `;
}
});
</script>

<user-card>
  <div>I like to swim.</div>
  <span slot="username">John Smith</span>
  <span slot="birthday">01.01.2001</span>
  <div>...And play volleyball too!</div>
</user-card>

```

Name: John Smith

Birthday: 01.01.2001

Other information

I like to swim.

...And play volleyball too!

All the unslotted light DOM content gets into the “Other information” fieldset.

Elements are appended to a slot one after another, so both unslotted pieces of information are in the default slot together.

The flattened DOM looks like this:

```

<user-card>
  #shadow-root
    <div>Name:
      <slot name="username">
        <span slot="username">John Smith</span>
      </slot>
    </div>
    <div>Birthday:
      <slot name="birthday">
        <span slot="birthday">01.01.2001</span>
      </slot>
    </div>
    <fieldset>
      <legend>About me</legend>
      <slot>

```

```

    <div>Hello</div>
    <div>I am John!</div>
  </slot>
</fieldset>
</user-card>

```

Menu example

Now let's back to `<custom-menu>`, mentioned at the beginning of the chapter.

We can use slots to distribute elements.

Here's the markup for `<custom-menu>`:

```

<custom-menu>
  <span slot="title">Candy menu</span>
  <li slot="item">Lollipop</li>
  <li slot="item">Fruit Toast</li>
  <li slot="item">Cup Cake</li>
</custom-menu>

```

The shadow DOM template with proper slots:

```

<template id="tpl">
  <style> /* menu styles */ </style>
  <div class="menu">
    <slot name="title"></slot>
    <ul><slot name="item"></slot></ul>
  </div>
</template>

```

1. `` goes into `<slot name="title">`.
2. There are many `<li slot="item">` in the template, but only one `<slot name="item">` in the template. That's perfectly normal. All elements with `slot="item"` get appended to `<slot name="item">` one after another, thus forming the list.

The flattened DOM becomes:

```

<custom-menu>
  #shadow-root
    <style> /* menu styles */ </style>
    <div class="menu">
      <slot name="title">
        <span slot="title">Candy menu</span>

```

```

    </slot>
  <ul>
    <slot name="item">
      <li slot="item">Lollipop</li>
      <li slot="item">Fruit Toast</li>
      <li slot="item">Cup Cake</li>
    </slot>
  </ul>
</div>
</custom-menu>

```

One might notice that, in a valid DOM, `` must be a direct child of ``. But that's flattened DOM, it describes how the component is rendered, such thing happens naturally here.

We just need to add a `click` handler to open/close the list, and the `<custom-menu>` is ready:

```

customElements.define('custom-menu', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});

    // tmpl is the shadow DOM template (above)
    this.shadowRoot.append( tmpl.content.cloneNode(true) );

    // we can't select light DOM nodes, so let's handle clicks on the slot
    this.shadowRoot.querySelector('slot[name="title"]').onclick = () => {
      // open/close the menu
      this.shadowRoot.querySelector('.menu').classList.toggle('closed');
    };
  }
});

```

Here's the full demo:

```

☐Candy menu
  Lollipop
  Fruit Toast
  Cup Cake

```

Of course, we can add more functionality to it: events, methods and so on.

Monitoring slots

What if the outer code wants to add/remove menu items dynamically?

The browser monitors slots and updates the rendering if slotted elements are added/removed.

Also, as light DOM nodes are not copied, but just rendered in slots, the changes inside them immediately become visible.

So we don't have to do anything to update rendering. But if the component wants to know about slot changes, then `slotchange` event is available.

For example, here the menu item is inserted dynamically after 1 second, and the title changes after 2 seconds:

```
<custom-menu id="menu">
  <span slot="title">Candy menu</span>
</custom-menu>

<script>
customElements.define('custom-menu', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<div class="menu">
      <slot name="title"></slot>
      <ul><slot name="item"></slot></ul>
    </div>`;

    // shadowRoot can't have event handlers, so using the first child
    this.shadowRoot.firstElementChild.addEventListener('slotchange',
      e => alert("slotchange: " + e.target.name)
    );
  }
});

setTimeout(() => {
  menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Lollipop</li>')
}, 1000);

setTimeout(() => {
  menu.querySelector('[slot="title"]').innerHTML = "New menu";
}, 2000);
</script>
```

The menu rendering updates each time without our intervention.

There are two `slotchange` events here:

1. At initialization:

`slotchange: title` triggers immediately, as the `slot="title"` from the light DOM gets into the corresponding slot.

2. After 1 second:

`slotchange`: `item` triggers, when a new `<li slot="item">` is added.

Please note: there's no `slotchange` event after 2 seconds, when the content of `slot="title"` is modified. That's because there's no slot change. We modify the content inside the slotted element, that's another thing.

If we'd like to track internal modifications of light DOM from JavaScript, that's also possible using a more generic mechanism: [MutationObserver](#).

Slot API

Finally, let's mention the slot-related JavaScript methods.

As we've seen before, JavaScript looks at the "real" DOM, without flattening. But, if the shadow tree has `{mode: 'open'}`, then we can figure out which elements assigned to a slot and, vice-versa, the slot by the element inside it:

- `node.assignedSlot` – returns the `<slot>` element that the `node` is assigned to.
- `slot.assignedNodes({flatten: true/false})` – DOM nodes, assigned to the slot. The `flatten` option is `false` by default. If explicitly set to `true`, then it looks more deeply into the flattened DOM, returning nested slots in case of nested components and the fallback content if no node assigned.
- `slot.assignedElements({flatten: true/false})` – DOM elements, assigned to the slot (same as above, but only element nodes).

These methods are useful when we need not just show the slotted content, but also track it in JavaScript.

For example, if `<custom-menu>` component wants to know, what it shows, then it could track `slotchange` and get the items from `slot.assignedElements`:

```
<custom-menu id="menu">
  <span slot="title">Candy menu</span>
  <li slot="item">Lollipop</li>
  <li slot="item">Fruit Toast</li>
</custom-menu>

<script>
customElements.define('custom-menu', class extends HTMLElement {
  items = []

  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<div class="menu">
      <slot name="title"></slot>
      <ul><slot name="item"></slot></ul>
    </div>`;
  }
});
```

```

// slottable is added/removed/replaced
this.shadowRoot.firstChild.addEventListener('slotchange', e => {
  let slot = e.target;
  if (slot.name == 'item') {
    this.items = slot.assignedElements().map(elem => elem.textContent);
    alert("Items: " + this.items);
  }
});

}
});

// items update after 1 second
setTimeout(() => {
  menu.insertAdjacentHTML('beforeEnd', '<li slot="item">Cup Cake</li>')
}, 1000);
</script>

```

Summary

Slots allow to show light DOM children in shadow DOM.

There are two kinds of slots:

- Named slots: `<slot name="X">...</slot>` – gets light children with `slot="X"`.
- Default slot: the first `<slot>` without a name (subsequent unnamed slots are ignored) – gets unslotted light children.
- If there are many elements for the same slot – they are appended one after another.
- The content of `<slot>` element is used as a fallback. It's shown if there are no light children for the slot.

The process of rendering slotted elements inside their slots is called “composition”. The result is called a “flattened DOM”.

Composition does not really move nodes, from JavaScript point of view the DOM is still same.

JavaScript can access slots using methods:

- `slot.assignedNodes/Elements()` – returns nodes/elements inside the slot.
- `node.assignedSlot` – the reverse method, returns slot by a node.

If we'd like to know what we're showing, we can track slot contents using:

- `slotchange` event – triggers the first time a slot is filled, and on any add/remove/replace operation of the slotted element, but not its children. The slot is `event.target`.
- [MutationObserver](#) to go deeper into slot content, watch changes inside it.

Now, as we have elements from light DOM in the shadow DOM, let's see how to style them properly. The basic rule is that shadow elements are styled inside, and light elements – outside, but there are notable exceptions.

We'll see the details in the next chapter.

Shadow DOM styling

Shadow DOM may include both `<style>` and `<link rel="stylesheet" href="...">` tags. In the latter case, stylesheets are HTTP-cached, so they are not redownloaded. There's no overhead in @importing or linking same styles for many components.

As a general rule, local styles work only inside the shadow tree, and document styles work outside of it. But there are few exceptions.

:host

The `:host` selector allows to select the shadow host (the element containing the shadow tree).

For instance, we're making `<custom-dialog>` element that should be centered. For that we need to style the `<custom-dialog>` element itself.

That's exactly what `:host` does:

```
<template id="tmpl">
  <style>
    /* the style will be applied from inside to the custom-dialog element */
    :host {
      position: fixed;
      left: 50%;
      top: 50%;
      transform: translate(-50%, -50%);
      display: inline-block;
      border: 1px solid red;
      padding: 10px;
    }
  </style>
  <slot></slot>
</template>

<script>
```

```

customElements.define('custom-dialog', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'}).append(tmpl.content.cloneNode(true));
  }
});
</script>

<custom-dialog>
  Hello!
</custom-dialog>

```

Hello!

Cascading

The shadow host (`<custom-dialog>` itself) resides in the light DOM, so it's affected by the main CSS cascade.

If there's a property styled both in `:host` locally, and in the document, then the document style takes precedence.

For instance, if in the document we had:

```

<style>
custom-dialog {
  padding: 0;
}
</style>

```

...Then the `<custom-dialog>` would be without padding.

It's very convenient, as we can setup "default" styles in the component `:host` rule, and then easily override them in the document.

The exception is when a local property is labelled `!important`, for such properties, local styles take precedence.

:host(selector)

Same as `:host`, but applied only if the shadow host matches the `selector`.

For example, we'd like to center the `<custom-dialog>` only if it has `centered` attribute:

```

<template id="tpl">
  <style>
    :host([centered]) {
      position: fixed;
      left: 50%;
      top: 50%;
      transform: translate(-50%, -50%);
    }

    :host {
      display: inline-block;
      border: 1px solid red;
      padding: 10px;
    }
  </style>
  <slot></slot>
</template>

<script>
customElements.define('custom-dialog', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'}).append(tmpl.content.cloneNode(true));
  }
});
</script>

<custom-dialog centered>
  Centered!
</custom-dialog>

<custom-dialog>
  Not centered.
</custom-dialog>

```

Not centered.

Centered!

Now the additional centering styles are only applied to the first dialog `<custom-dialog centered>`.

`:host-context(selector)`

Same as `:host`, but applied only if the shadow host or any of its ancestors in the outer document matches the `selector`.

E.g. `:host-context(.dark-theme)` matches only if there's `dark-theme` class on `<custom-dialog>` on above it:

```

<body class="dark-theme">
  <!--
    :host-context(.dark-theme) applies to custom-dialogs inside .dark-theme
  -->
  <custom-dialog>...</custom-dialog>
</body>

```

To summarize, we can use `:host` -family of selectors to style the main element of the component, depending on the context. These styles (unless `!important`) can be overridden by the document.

Styling slotted content

Now let's consider the situation with slots.

Slotted elements come from light DOM, so they use document styles. Local styles do not affect slotted content.

In the example below, slotted `` is bold, as per document style, but does not take `background` from the local style:

```

<style>
  span { font-weight: bold }
</style>

<user-card>
  <div slot="username"><span>John Smith</span></div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <style>
        span { background: red; }
      </style>
      Name: <slot name="username"></slot>
    `;
  }
});
</script>

```

Name:
John Smith

The result is bold, but not red.

If we'd like to style slotted elements in our component, there are two choices.

First, we can style the `<slot>` itself and rely on CSS inheritance:

```
<user-card>
  <div slot="username"><span>John Smith</span></div>
</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <style>
        slot[name="username"] { font-weight: bold; }
      </style>
      Name: <slot name="username"></slot>
    `;
  }
});
</script>
```

Name:
John Smith

Here `<p>John Smith</p>` becomes bold, because CSS inheritance is in effect between the `<slot>` and its contents. But not all CSS properties are inherited.

Another option is to use `::slotted(selector)` pseudo-class. It matches elements based on two conditions:

1. The element from the light DOM that is inserted into a `<slot>`. Then slot name doesn't matter. Just any slotted element, but only the element itself, not its children.
2. The element matches the `selector`.

In our example, `::slotted(div)` selects exactly `<div slot="username">`, but not its children:

```
<user-card>
  <div slot="username">
    <div>John Smith</div>
  </div>
</user-card>
```

```

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `
      <style>
        ::slotted(div) { border: 1px solid red; }
      </style>
      Name: <slot name="username"></slot>
    `;
  }
});
</script>

```

Name:

John Smith

Please note, `::slotted` selector can't descend any further into the slot. These selectors are invalid:

```

::slotted(div span) {
  /* our slotted <div> does not match this */
}

::slotted(div) p {
  /* can't go inside light DOM */
}

```

Also, `::slotted` can only be used in CSS. We can't use it in `querySelector`.

CSS hooks with custom properties

How do we style a component in-depth from the main document?

Naturally, document styles apply to `<custom-dialog>` element or `<user-card>`, etc. But how can we affect its internals? For instance, in `<user-card>` we'd like to allow the outer document change how user fields look.

Just as we expose methods to interact with our component, we can expose CSS variables (custom CSS properties) to style it.

Custom CSS properties exist on all levels, both in light and shadow.

For example, in shadow DOM we can use `--user-card-field-color` CSS variable to style fields:


```

<style>
  .field {
    color: var(--user-card-field-color, black);
    /* if --user-card-field-color is not defined, use black */
  }
</style>
<div class="field">Name: <slot name="username"></slot></div>
<div class="field">Birthday: <slot name="birthday"></slot></div>
</style>

```

Then, we can declare this property in the outer document for `<user-card>`:

```

user-card {
  --user-card-field-color: green;
}

```

Custom CSS properties pierce through shadow DOM, they are visible everywhere, so the inner `.field` rule will make use of it.

Here's the full example:

```

<style>
  user-card {
    --user-card-field-color: green;
  }
</style>

<template id="tpl">
  <style>
    .field {
      color: var(--user-card-field-color, black);
    }
  </style>
  <div class="field">Name: <slot name="username"></slot></div>
  <div class="field">Birthday: <slot name="birthday"></slot></div>
</template>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.append(document.getElementById('tpl').content.cloneNode(true))
  }
});
</script>

<user-card>
  <span slot="username">John Smith</span>

```

```
<span slot="birthday">01.01.2001</span>  
</user-card>
```

Name: John Smith
Birthday: 01.01.2001

Summary

Shadow DOM can include styles, such as `<style>` or `<link rel="stylesheet">`.

Local styles can affect:

- shadow tree,
- shadow host with `:host` -family pseudoclasses,
- slotted elements (coming from light DOM), `::slotted(selector)` allows to select slotted elements themselves, but not their children.

Document styles can affect:

- shadow host (as it's in the outer document)
- slotted elements and their contents (as it's physically in the outer document)

When CSS properties conflict, normally document styles have precedence, unless the property is labelled as `!important`. Then local styles have precedence.

CSS custom properties pierce through shadow DOM. They are used as “hooks” to style the component:

1. The component uses a custom CSS property to style key elements, such as `var(--component-name-title, <default value>)`.
2. Component author publishes these properties for developers, they are same important as other public component methods.
3. When a developer wants to style a title, they assign `--component-name-title` CSS property for the shadow host or above.
4. Profit!

Shadow DOM and events

The idea behind shadow tree is to encapsulate internal implementation details of a component.

Let's say, a click event happens inside a shadow DOM of `<user-card>` component. But scripts in the main document have no idea about the shadow DOM internals, especially if the component comes from a 3rd-party library.

So, to keep the details encapsulated, the browser *retargets* the event.

Events that happen in shadow DOM have the host element as the target, when caught outside of the component.

Here's a simple example:

```
<user-card></user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<p>
      <button>Click me</button>
    </p>`;
    this.shadowRoot.firstElementChild.onclick =
      e => alert("Inner target: " + e.target.tagName);
  }
});

document.onclick =
  e => alert("Outer target: " + e.target.tagName);
</script>
```

Click me

If you click on the button, the messages are:

1. Inner target: **BUTTON** – internal event handler gets the correct target, the element inside shadow DOM.
2. Outer target: **USER-CARD** – document event handler gets shadow host as the target.

Event retargeting is a great thing to have, because the outer document doesn't have no know about component internals. From its point of view, the event happened on **<user-card>**.

Retargeting does not occur if the event occurs on a slotted element, that physically lives in the light DOM.

For example, if a user clicks on **** in the example below, the event target is exactly this **span** element, for both shadow and light handlers:

```
<user-card id="userCard">
  <span slot="username">John Smith</span>
```

```

</user-card>

<script>
customElements.define('user-card', class extends HTMLElement {
  connectedCallback() {
    this.attachShadow({mode: 'open'});
    this.shadowRoot.innerHTML = `<div>
      <b>Name:</b> <slot name="username"></slot>
    </div>`;

    this.shadowRoot.firstElementChild.onclick =
      e => alert("Inner target: " + e.target.tagName);
  }
});

userCard.onclick = e => alert(`Outer target: ${e.target.tagName}`);
</script>

```

Name: John Smith

If a click happens on "John Smith", for both inner and outer handlers the target is ``. That's an element from the light DOM, so no retargeting.

On the other hand, if the click occurs on an element originating from shadow DOM, e.g. on `Name`, then, as it bubbles out of the shadow DOM, its `event.target` is reset to `<user-card>`.

Bubbling, `event.composedPath()`

For purposes of event bubbling, flattened DOM is used.

So, if we have a slotted element, and an event occurs somewhere inside it, then it bubbles up to the `<slot>` and upwards.

The full path to the original event target, with all the shadow elements, can be obtained using `event.composedPath()`. As we can see from the name of the method, that path is taken after the composition.

In the example above, the flattened DOM is:

```

<user-card id="userCard">
  #shadow-root
    <div>
      <b>Name:</b>
      <slot name="username">
        <span slot="username">John Smith</span>
      </slot>
    </div>
  </user-card>

```

```
</div>  
</user-card>
```

So, for a click on ``, a call to `event.composedPath()` returns an array: `[span, slot, div, shadow-root, user-card, body, html, document, window]`. That's exactly the parent chain from the target element in the flattened DOM, after the composition.

⚠ Shadow tree details are only provided for `{mode: 'open'}` trees

If the shadow tree was created with `{mode: 'closed'}`, then the composed path starts from the host: `user-card` and upwards.

That's the similar principle as for other methods that work with shadow DOM. Internals of closed trees are completely hidden.

event.composed

Most events successfully bubble through a shadow DOM boundary. There are few events that do not.

This is governed by the `composed` event object property. If it's `true`, then the event does cross the boundary. Otherwise, it only can be caught from inside the shadow DOM.

If you take a look at [UI Events specification](#), most events have `composed: true`:

- `blur`, `focus`, `focusin`, `focusout`,
- `click`, `dblclick`,
- `mousedown`, `mouseup`, `mousemove`, `mouseout`, `mouseover`,
- `wheel`,
- `beforeinput`, `input`, `keydown`, `keyup`.

All touch events and pointer events also have `composed: true`.

There are some events that have `composed: false` though:

- `mouseenter`, `mouseleave` (they do not bubble at all),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

These events can be caught only on elements within the same DOM, where the event target resides.

Custom events

When we dispatch custom events, we need to set both `bubbles` and `composed` properties to `true` for it to bubble up and out of the component.

For example, here we create `div#inner` in the shadow DOM of `div#outer` and trigger two events on it. Only the one with `composed: true` makes it outside to the document:

```
<div id="outer"></div>

<script>
outer.attachShadow({mode: 'open'});

let inner = document.createElement('div');
outer.shadowRoot.append(inner);

/*
div(id=outer)
  #shadow-dom
    div(id=inner)
*/

document.addEventListener('test', event => alert(event.detail));

inner.dispatchEvent(new CustomEvent('test', {
  bubbles: true,
  composed: true,
  detail: "composed"
}));

inner.dispatchEvent(new CustomEvent('test', {
  bubbles: true,
  composed: false,
  detail: "not composed"
}));
</script>
```

Summary

Events only cross shadow DOM boundaries if their `composed` flag is set to `true`.

Built-in events mostly have `composed: true`, as described in the relevant specifications:

- UI Events <https://www.w3.org/TR/uievents>.

- Touch Events <https://w3c.github.io/touch-events> .
- Pointer Events <https://www.w3.org/TR/pointerevents> .
- ...And so on.

Some built-in events that have `composed: false`:

- `mouseenter`, `mouseleave` (also do not bubble),
- `load`, `unload`, `abort`, `error`,
- `select`,
- `slotchange`.

These events can be caught only on elements within the same DOM.

If we dispatch a `CustomEvent`, then we should explicitly set `composed: true`.

Please note that in case of nested components, one shadow DOM may be nested into another. In that case composed events bubble through all shadow DOM boundaries. So, if an event is intended only for the immediate enclosing component, we can also dispatch it on the shadow host and set `composed: false`. Then it's out of the component shadow DOM, but won't bubble up to higher-level DOM.

Regular expressions

Regular expressions is a powerful way of doing search and replace in strings.

In JavaScript regular expressions are implemented using objects of a built-in `RegExp` class and integrated with strings.

Please note that regular expressions vary between programming languages. In this tutorial we concentrate on JavaScript. Of course there's a lot in common, but they are a somewhat different in Perl, Ruby, PHP etc.

Patterns and flags

A regular expression (also “regex”, or just “reg”) consists of a *pattern* and optional *flags*.

There are two syntaxes to create a regular expression object.

The long syntax:


```
regex = new RegExp("pattern", "flags");
```

...And the short one, using slashes `"/"`:

```
regexp = /pattern/; // no flags
regexp = /pattern/gmi; // with flags g,m and i (to be covered soon)
```

Slashes `"/"` tell JavaScript that we are creating a regular expression. They play the same role as quotes for strings.

Usage

To search inside a string, we can use method [search](#) .

Here's an example:

```
let str = "I love JavaScript!"; // will search here

let regexp = /love/;
alert( str.search(regexp) ); // 2
```

The `str.search` method looks for the pattern `/love/` and returns the position inside the string. As we might guess, `/love/` is the simplest possible pattern. What it does is a simple substring search.

The code above is the same as:

```
let str = "I love JavaScript!"; // will search here

let substr = 'love';
alert( str.search(substr) ); // 2
```

So searching for `/love/` is the same as searching for `"love"`.

But that's only for now. Soon we'll create more complex regular expressions with much more searching power.

Colors

From here on the color scheme is:

- `regexp` – red
- `string` (where we search) – blue
- `result` – green

i When to use `new RegExp` ?

Normally we use the short syntax `/.../`. But it does not support variable insertions `${...}`.

On the other hand, `new RegExp` allows to construct a pattern dynamically from a string, so it's more flexible.

Here's an example of a dynamically generated regexp:

```
let tag = prompt("Which tag you want to search?", "h2");
let regexp = new RegExp(`<${tag}>`);

// finds <h2> by default
alert( "<h1> <h2> <h3>".search(regexp));
```

Flags

Regular expressions may have flags that affect the search.

There are only 6 of them in JavaScript:

i

With this flag the search is case-insensitive: no difference between `A` and `a` (see the example below).

g

With this flag the search looks for all matches, without it – only the first one (we'll see uses in the next chapter).

m

Multiline mode (covered in the chapter [Multiline mode, flag "m"](#)).

s

“Dotall” mode, allows `.` to match newlines (covered in the chapter [Character classes](#)).

u

Enables full unicode support. The flag enables correct processing of surrogate pairs. More about that in the chapter [Unicode: flag "u"](#).

y

Sticky mode (covered in the chapter [Sticky flag "y", searching at position](#))

We'll cover all these flags further in the tutorial.

For now, the simplest flag is `i`, here's an example:

```
let str = "I love JavaScript!";

alert( str.search(/LOVE/i) ); // 2 (found lowercased)

alert( str.search(/LOVE/) ); // -1 (nothing found without 'i' flag)
```

So the `i` flag already makes regular expressions more powerful than a simple substring search. But there's so much more. We'll cover other flags and features in the next chapters.

Summary

- A regular expression consists of a pattern and optional flags: `g`, `i`, `m`, `u`, `s`, `y`.
- Without flags and special symbols that we'll study later, the search by a regexp is the same as a substring search.
- The method `str.search(regexp)` returns the index where the match is found or `-1` if there's no match. In the next chapter we'll see other methods.

Methods of RegExp and String

There are two sets of methods to deal with regular expressions.

1. First, regular expressions are objects of the built-in [RegExp](#) class, it provides many methods.
2. Besides that, there are methods in regular strings can work with regexps.

Recipes

Which method to use depends on what we'd like to do.

Methods become much easier to understand if we separate them by their use in real-life tasks.

So, here are general recipes, the details to follow:

To search for all matches:

Use regexp `g` flag and:

- Get a flat array of matches – `str.match(reg)`
- Get an array of matches with details – `str.matchAll(reg)`.

To search for the first match only:

- Get the full first match – `str.match(reg)` (without `g` flag).
- Get the string position of the first match – `str.search(reg)`.
- Check if there's a match – `regex.test(str)`.
- Find the match from the given position – `regex.exec(str)` (set `regex.lastIndex` to position).

To replace all matches:

- Replace with another string or a function result – `str.replace(reg, str|func)`

To split the string by a separator:

- `str.split(str|reg)`

Now you can continue reading this chapter to get the details about every method... But if you're reading for the first time, then you probably want to know more about regexps. So you can move to the next chapter, and then return here if something about a method is unclear.

`str.search(reg)`

We've seen this method already. It returns the position of the first match or `-1` if none found:

```
let str = "A drop of ink may make a million think";

alert( str.search( /a/i ) ); // 0 (first match at zero position)
```

The important limitation: `search` only finds the first match.

We can't find next matches using `search`, there's just no syntax for that. But there are other methods that can.

`str.match(reg)`, no “g” flag

The behavior of `str.match` varies depending on whether `reg` has `g` flag or not. First, if there's no `g` flag, then `str.match(reg)` looks for the first match only.

The result is an array with that match and additional properties:

- `index` – the position of the match inside the string,
- `input` – the subject string.

For instance:

```
let str = "Fame is the thirst of youth";

let result = str.match( /fame/i );

alert( result[0] );    // Fame (the match)
alert( result.index ); // 0 (at the zero position)
alert( result.input ); // "Fame is the thirst of youth" (the string)
```

A match result may have more than one element.

If a part of the pattern is delimited by parentheses (. . .), then it becomes a separate element in the array.

If parentheses have a name, designated by (?<name> . . .) at their start, then `result.groups[name]` has the content. We'll see that later in the chapter [about groups](#).

For instance:

```
let str = "JavaScript is a programming language";

let result = str.match( /JAVA(SCRIPT)/i );

alert( result[0] ); // JavaScript (the whole match)
alert( result[1] ); // script (the part of the match that corresponds to the parent)
alert( result.index ); // 0
alert( result.input ); // JavaScript is a programming language
```

Due to the `i` flag the search is case-insensitive, so it finds JavaScript. The part of the match that corresponds to SCRIPT becomes a separate array item.

So, this method is used to find one full match with all details.

str.match(reg) with “g” flag

When there's a `"g"` flag, then `str.match` returns an array of all matches. There are no additional properties in that array, and parentheses do not create any elements.

For instance:

```
let str = "H0-Ho-ho!";  
  
let result = str.match( /ho/ig );  
  
alert( result ); // H0, Ho, ho (array of 3 matches, case-insensitive)
```

Parentheses do not change anything, here we go:

```
let str = "H0-Ho-ho!";  
  
let result = str.match( /h(o)/ig );  
  
alert( result ); // H0, Ho, ho
```

So, with `g` flag `str.match` returns a simple array of all matches, without details.

If we want to get information about match positions and contents of parentheses then we should use `matchAll` method that we'll cover below.

⚠ If there are no matches, `str.match` returns `null`

Please note, that's important. If there are no matches, the result is not an empty array, but `null`.

Keep that in mind to evade pitfalls like this:

```
let str = "Hey-hey-hey!";  
  
alert( str.match(/Z/g).length ); // Error: Cannot read property 'length' of null
```

Here `str.match(/Z/g)` is `null`, it has no `length` property.

`str.matchAll(regex)`

The method `str.matchAll(regex)` is used to find all matches with all details.

For instance:

```
let str = "Javascript or JavaScript? Should we uppercase 'S'?";
```

```
let result = str.matchAll( /java(script)/ig );

let [match1, match2] = result;

alert( match1[0] ); // Javascript (the whole match)
alert( match1[1] ); // script (the part of the match that corresponds to the parent)
alert( match1.index ); // 0
alert( match1.input ); // = str (the whole original string)

alert( match2[0] ); // JavaScript (the whole match)
alert( match2[1] ); // Script (the part of the match that corresponds to the parent)
alert( match2.index ); // 14
alert( match2.input ); // = str (the whole original string)
```

⚠ `matchAll` returns an iterable, not array

For instance, if we try to get the first match by index, it won't work:

```
let str = "Javascript or JavaScript?";

let result = str.matchAll( /javascript/ig );

alert(result[0]); // undefined (! there must be a match)
```

The reason is that the iterator is not an array. We need to run `Array.from(result)` on it, or use `for...of` loop to get matches.

In practice, if we need all matches, then `for...of` works, so it's not a problem.

And, to get only few matches, we can use destructuring:

```
let str = "Javascript or JavaScript?";

let [firstMatch] = str.matchAll( /javascript/ig );

alert(firstMatch); // Javascript
```

⚠ `matchAll` is supernew, may need a polyfill

The method may not work in old browsers. A polyfill might be needed (this site uses core-js).

Or you could make a loop with `regexp.exec`, explained below.

`str.split(regex|substr, limit)`

Splits the string using the regexp (or a substring) as a delimiter.

We already used `split` with strings, like this:

```
alert('12-34-56'.split('-')) // array of [12, 34, 56]
```

But we can split by a regular expression, the same way:

```
alert('12-34-56'.split(/-/)) // array of [12, 34, 56]
```

str.replace(str|reg, str|func)

This is a generic method for searching and replacing, one of most useful ones. The swiss army knife for searching and replacing.

We can use it without regexps, to search and replace a substring:

```
// replace a dash by a colon
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

There's a pitfall though.

When the first argument of `replace` is a string, it only looks for the first match.

You can see that in the example above: only the first `" - "` is replaced by `" : "`.

To find all dashes, we need to use not the string `" - "`, but a regexp `/-/g`, with an obligatory `g` flag:

```
// replace all dashes by a colon
alert('12-34-56'.replace(/-/g, ":")) // 12:34:56
```

The second argument is a replacement string. We can use special characters in it:

Symbol	Inserts
<code>\$\$</code>	<code>"\$"</code>
<code>\$&</code>	the whole match
<code>\$`</code>	a part of the string before the match
<code>\$'</code>	a part of the string after the match

Symbol	Inserts
--------	---------

\$n	if n is a 1-2 digit number, then it means the contents of n-th parentheses counting from left to right, otherwise it means a parentheses with the given name
-----	---

For instance if we use `$&` in the replacement string, that means “put the whole match here”.

Let’s use it to prepend all entries of `"John"` with `"Mr. "`:

```
let str = "John Doe, John Smith and John Bull";  
  
// for each John - replace it with Mr. and then John  
alert(str.replace(/John/g, 'Mr.$&')); // Mr.John Doe, Mr.John Smith and Mr.John Bu.
```

Quite often we’d like to reuse parts of the source string, recombine them in the replacement or wrap into something.

To do so, we should:

1. First, mark the parts by parentheses in regexp.
2. Use `$1`, `$2` (and so on) in the replacement string to get the content matched by 1st, 2nd and so on parentheses.

For instance:

```
let str = "John Smith";  
  
// swap first and last name  
alert(str.replace(/(john) (smith)/i, '$2, $1')) // Smith, John
```

For situations that require “smart” replacements, the second argument can be a function.

It will be called for each match, and its result will be inserted as a replacement.

For instance:

```
let i = 0;  
  
// replace each "ho" by the result of the function  
alert("HO-Ho-ho".replace(/ho/gi, function() {  
    return ++i;  
})); // 1-2-3
```


In the example above the function just returns the next number every time, but usually the result is based on the match.

The function is called with arguments `func(str, p1, p2, ..., pn, offset, input, groups)`:

1. `str` – the match,
2. `p1, p2, ..., pn` – contents of parentheses (if there are any),
3. `offset` – position of the match,
4. `input` – the source string,
5. `groups` – an object with named groups (see chapter [Capturing groups](#)).

If there are no parentheses in the regexp, then there are only 3 arguments: `func(str, offset, input)`.

Let's use it to show full information about matches:

```
// show and replace all matches
function replacer(str, offset, input) {
  alert(`Found ${str} at position ${offset} in string ${input}`);
  return str.toLowerCase();
}

let result = "H0-Ho-ho".replace(/ho/gi, replacer);
alert( 'Result: ' + result ); // Result: ho-ho-ho

// shows each match:
// Found H0 at position 0 in string H0-Ho-ho
// Found Ho at position 3 in string H0-Ho-ho
// Found ho at position 6 in string H0-Ho-ho
```

In the example below there are two parentheses, so `replacer` is called with 5 arguments: `str` is the full match, then parentheses, and then `offset` and `input`:

```
function replacer(str, name, surname, offset, input) {
  // name is the first parentheses, surname is the second one
  return surname + ", " + name;
}

let str = "John Smith";

alert(str.replace(/(John) (Smith)/, replacer)) // Smith, John
```

Using a function gives us the ultimate replacement power, because it gets all the information about the match, has access to outer variables and can do everything.

regexp.exec(str)

We've already seen these searching methods:

- `search` – looks for the position of the match,
- `match` – if there's no `g` flag, returns the first match with parentheses and all details,
- `match` – if there's a `g` flag – returns all matches, without details parentheses,
- `matchAll` – returns all matches with details.

The `regexp.exec` method is the most flexible searching method of all. Unlike previous methods, `exec` should be called on a regexp, rather than on a string.

It behaves differently depending on whether the regexp has the `g` flag.

If there's no `g`, then `regexp.exec(str)` returns the first match, exactly as `str.match(reg)`. Such behavior does not give us anything new.

But if there's `g`, then:

- `regexp.exec(str)` returns the first match and *remembers* the position after it in `regexp.lastIndex` property.
- The next call starts to search from `regexp.lastIndex` and returns the next match.
- If there are no more matches then `regexp.exec` returns `null` and `regexp.lastIndex` is set to `0`.

We could use it to get all matches with their positions and parentheses groups in a loop, instead of `matchAll`:

```
let str = 'A lot about JavaScript at https://javascript.info';

let regexp = /javascript/ig;

let result;

while (result = regexp.exec(str)) {
  alert( `Found ${result[0]} at ${result.index}` );
  // shows: Found JavaScript at 12, then:
  // shows: Found javascript at 34
}
```

Surely, `matchAll` does the same, at least for modern browsers. But what `matchAll` can't do – is to search from a given position.

Let's search from position `13`. What we need is to assign `regex.lastIndex=13` and call `regex.exec`:

```
let str = "A lot about JavaScript at https://javascript.info";

let regex = /javascript/ig;
regex.lastIndex = 13;

let result;

while (result = regex.exec(str)) {
  alert( `Found ${result[0]} at ${result.index}` );
  // shows: Found javascript at 34
}
```

Now, starting from the given position `13`, there's only one match.

`regex.test(str)`

The method `regex.test(str)` looks for a match and returns `true/false` whether it finds it.

For instance:

```
let str = "I love JavaScript";

// these two tests do the same
alert( /love/i.test(str) ); // true
alert( str.search(/love/i) !== -1 ); // true
```

An example with the negative answer:

```
let str = "Bla-bla-bla";

alert( /love/i.test(str) ); // false
alert( str.search(/love/i) !== -1 ); // false
```

If the `regex` has `'g'` flag, then `regex.test` advances `regex.lastIndex` property, just like `regex.exec`.

So we can use it to search from a given position:

```
let regexp = /love/gi;

let str = "I love JavaScript";

// start the search from position 10:
regexp.lastIndex = 10
alert( regexp.test(str) ); // false (no match)
```

⚠ Same global regexp tested repeatedly may fail to match

If we apply the same global regexp to different inputs, it may lead to wrong result, because `regexp.test` call advances `regexp.lastIndex` property, so the search in another string may start from non-zero position.

For instance, here we call `regexp.test` twice on the same text, and the second time fails:

```
let regexp = /javascript/g; // (regexp just created: regexp.lastIndex=0)

alert( regexp.test("javascript") ); // true (regexp.lastIndex=10 now)
alert( regexp.test("javascript") ); // false
```

That's exactly because `regexp.lastIndex` is non-zero on the second test.

To work around that, one could use non-global regexps or re-adjust `regexp.lastIndex=0` before a new search.

Summary

There's a variety of many methods on both regexps and strings.

Their abilities and methods overlap quite a bit, we can do the same by different calls. Sometimes that may cause confusion when starting to learn the language.

Then please refer to the recipes at the beginning of this chapter, as they provide solutions for the majority of regexp-related tasks.

Character classes

Consider a practical task – we have a phone number `" +7(903)-123-45-67 "`, and we need to turn it into pure numbers: `79035419441`.

To do so, we can find and remove anything that's not a number. Character classes can help with that.

A character class is a special notation that matches any symbol from a certain set.

For the start, let's explore a "digit" class. It's written as `\d`. We put it in the pattern, that means "any single digit".

For instance, let's find the first digit in the phone number:

```
let str = "+7(903)-123-45-67";  
  
let reg = /\d/;  
  
alert( str.match(reg) ); // 7
```

Without the flag `g`, the regular expression only looks for the first match, that is the first digit `\d`.

Let's add the `g` flag to find all digits:

```
let str = "+7(903)-123-45-67";  
  
let reg = /\d/g;  
  
alert( str.match(reg) ); // array of matches: 7,9,0,3,1,2,3,4,5,6,7  
  
alert( str.match(reg).join('') ); // 79035419441
```

That was a character class for digits. There are other character classes as well.

Most used are:

`\d` ("d" is from "digit")

A digit: a character from `0` to `9`.

`\s` ("s" is from "space")

A space symbol: that includes spaces, tabs, newlines.

`\w` ("w" is from "word")

A "wordly" character: either a letter of English alphabet or a digit or an underscore. Non-Latin letters (like cyrillic or hindi) do not belong to `\w`.

For instance, `\d\s\w` means a "digit" followed by a "space character" followed by a "wordly character", like `"1 a"`.

A regexp may contain both regular symbols and character classes.

For instance, `CSS\d` matches a string `CSS` with a digit after it:

```
let str = "CSS4 is cool";
let reg = /CSS\d/

alert( str.match(reg) ); // CSS4
```

Also we can use many character classes:

```
alert( "I love HTML5!".match(/s\w\w\w\w\d/) ); // ' HTML5'
```

The match (each character class corresponds to one result character):

I love HTML5

(The image shows the string "I love HTML5" with a red bracket above "love" spanning the characters 'l', 'o', 'v', 'e', ' ', 'H', 'T', 'M', 'L', '5'. The characters 'l', 'o', 'v', 'e', ' ', 'H', 'T', 'M', 'L', '5' are highlighted in blue.)

Word boundary: \b

A word boundary \b – is a special character class.

It does not denote a character, but rather a boundary between characters.

For instance, \bJava\b matches Java in the string Hello, Java!, but not in the script Hello, JavaScript!.

```
alert( "Hello, Java!".match(/bJava\b/) ); // Java
alert( "Hello, JavaScript!".match(/bJava\b/) ); // null
```

The boundary has “zero width” in a sense that usually a character class means a character in the result (like a wordly character or a digit), but not in this case.

The boundary is a test.

When regular expression engine is doing the search, it's moving along the string in an attempt to find the match. At each string position it tries to find the pattern.

When the pattern contains \b, it tests that the position in string is a word boundary, that is one of three variants:

- Immediately before is \w, and immediately after – not \w, or vise versa.
- At string start, and the first string character is \w.
- At string end, and the last string character is \w.

For instance, in the string Hello, Java! the following positions match \b:

↓ ↓ ↓ ↓
Hello, Java!

So it matches \bHello\b, because:

1. At the beginning of the string the first \b test matches.
2. Then the word Hello matches.
3. Then \b matches, as we're between o and a space.

Pattern \bJava\b also matches. But not \bHell\b (because there's no word boundary after l) and not Java!\b (because the exclamation sign is not a wordly character, so there's no word boundary after it).

```
alert( "Hello, Java!".match(/^\bHello\b/) ); // Hello
alert( "Hello, Java!".match(/^\bJava\b/) ); // Java
alert( "Hello, Java!".match(/^\bHell\b/) ); // null (no match)
alert( "Hello, Java!".match(/^\bJava!\b/) ); // null (no match)
```

Once again let's note that \b makes the searching engine to test for the boundary, so that Java\b finds Java only when followed by a word boundary, but it does not add a letter to the result.

Usually we use \b to find standalone English words. So that if we want "Java" language then \bJava\b finds exactly a standalone word and ignores it when it's a part of another word, e.g. it won't match Java in JavaScript.

Another example: a regexp \b\d\d\b looks for standalone two-digit numbers. In other words, it requires that before and after \d\d must be a symbol different from \w (or beginning/end of the string).

```
alert( "1 23 456 78".match(/^\b\d\d\b/g) ); // 23,78
```

⚠ Word boundary doesn't work for non-Latin alphabets

The word boundary check \b tests for a boundary between \w and something else. But \w means an English letter (or a digit or an underscore), so the test won't work for other characters (like cyrillic or hieroglyphs).

Later we'll come by Unicode character classes that allow to solve the similar task for different languages.

Inverse classes

For every character class there exists an “inverse class”, denoted with the same letter, but uppercased.

The “reverse” means that it matches all other characters, for instance:

\D

Non-digit: any character except **\d** , for instance a letter.

\S

Non-space: any character except **\s** , for instance a letter.

\W

Non-wordly character: anything but **\w** .

\B

Non-boundary: a test reverse to **\b** .

In the beginning of the chapter we saw how to get all digits from the phone +7(903)-123-45-67 .

One way was to match all digits and join them:

```
let str = "+7(903)-123-45-67";  
alert( str.match(/\d/g).join('') ); // 79031234567
```

An alternative, shorter way is to find non-digits **\D** and remove them from the string:

```
let str = "+7(903)-123-45-67";  
alert( str.replace(/\D/g, '') ); // 79031234567
```

Spaces are regular characters

Usually we pay little attention to spaces. For us strings 1-5 and 1 - 5 are nearly identical.

But if a regexp doesn't take spaces into account, it may fail to work.

Let's try to find digits separated by a dash:

```
alert( "1 - 5".match(/\d-\d/) ); // null, no match!
```


Here we fix it by adding spaces into the regexp `\d - \d`:

```
alert( "1 - 5".match(/ \d - \d/ ) ); // 1 - 5, now it works
```

A space is a character. Equal in importance with any other character.

Of course, spaces in a regexp are needed only if we look for them. Extra spaces (just like any other extra characters) may prevent a match:

```
alert( "1-5".match(/ \d - \d/ ) ); // null, because the string 1-5 has no spaces
```

In other words, in a regular expression all characters matter, spaces too.

A dot is any character

The dot `"."` is a special character class that matches “any character except a newline”.

For instance:

```
alert( "Z".match(/./) ); // Z
```

Or in the middle of a regexp:

```
let reg = /CS.4/;

alert( "CSS4".match(reg) ); // CSS4
alert( "CS-4".match(reg) ); // CS-4
alert( "CS 4".match(reg) ); // CS 4 (space is also a character)
```

Please note that the dot means “any character”, but not the “absense of a character”. There must be a character to match it:

```
alert( "CS4".match(/CS.4/) ); // null, no match because there's no character for the dot
```

The dotall “s” flag

Usually a dot doesn’t match a newline character.

For instance, `A.B` matches `A`, and then `B` with any character between them, except a newline.

This doesn't match:

```
alert( "A\nB".match(/A.B/) ); // null (no match)

// a space character would match, or a letter, but not \n
```

Sometimes it's inconvenient, we really want “any character”, newline included.

That's what `s` flag does. If a regexp has it, then the dot `.` match literally any character:

```
alert( "A\nB".match(/A.B/s) ); // A\nB (match!)
```

Summary

There exist following character classes:

- `\d` – digits.
- `\D` – non-digits.
- `\s` – space symbols, tabs, newlines.
- `\S` – all but `\s`.
- `\w` – English letters, digits, underscore `'_'`.
- `\W` – all but `\w`.
- `.` – any character if with the regexp `'s'` flag, otherwise any except a newline.

...But that's not all!

The Unicode encoding, used by JavaScript for strings, provides many properties for characters, like: which language the letter belongs to (if a letter) it is it a punctuation sign, etc.

Modern JavaScript allows to use these properties in regexps to look for characters, for instance:

- A cyrillic letter is: `\p{Script=Cyrillic}` or `\p{sc=Cyrillic}`.
- A dash (be it a small hyphen `-` or a long dash `—`): `\p{Dash_Punctuation}` or `\p{pd}`.
- A currency symbol, such as `$`, `€` or another: `\p{Currency_Symbol}` or `\p{sc}`.
- ...And much more. Unicode has a lot of character categories that we can select from.

These patterns require `'u'` regexp flag to work. More about that in the chapter [Unicode: flag "u"](#).

✓ Tasks

Find the time

The time has a format: `hours:minutes`. Both hours and minutes has two digits, like `09:00`.

Make a regexp to find time in the string: Breakfast at 09:00 in the room 123:456.

P.S. In this task there's no need to check time correctness yet, so `25:99` can also be a valid result. P.P.S. The regexp shouldn't match `123:456`.

[To solution](#)

Escaping, special characters

As we've seen, a backslash `"\"` is used to denote character classes. So it's a special character in regexps (just like in a regular string).

There are other special characters as well, that have special meaning in a regexp.

They are used to do more powerful searches. Here's a full list of them: `[\ ^ $. | ? * + ()`.

Don't try to remember the list – soon we'll deal with each of them separately and you'll know them by heart automatically.

Escaping

Let's say we want to find a dot literally. Not “any character”, but just a dot.

To use a special character as a regular one, prepend it with a backslash: `\.`

That's also called “escaping a character”.

For example:

```
alert( "Chapter 5.1".match(/\\d\\.\\d/) ); // 5.1 (match!)
alert( "Chapter 511".match(/\\d\\.\\d/) ); // null (looking for a real dot \\.)
```

Parentheses are also special characters, so if we want them, we should use \(. The example below looks for a string "g()":

```
alert( "function g()".match(/g\(\)/) ); // "g()"
```

If we're looking for a backslash `\`, it's a special character in both regular strings and regexps, so we should double it.

```
alert( "1\\2".match(/\\/) ); // '\'
```

A slash

A slash symbol `'/'` is not a special character, but in JavaScript it is used to open and close the regexp: /...pattern.../, so we should escape it too.

Here's what a search for a slash `'/'` looks like:

```
alert( "/" .match(/\//) ); // '/'
```

On the other hand, if we're not using `/.../`, but create a regexp using `new RegExp`, then we don't need to escape it:

```
alert( "/" .match(new RegExp("/")) ); // '/'
```

new RegExp

If we are creating a regular expression with `new RegExp`, then we don't have to escape `/`, but need to do some other escaping.

For instance, consider this:

```
let reg = new RegExp("\d\.\d");  
  
alert( "Chapter 5.1".match(reg) ); // null
```

The search worked with /\d\.\d/, but with `new RegExp("\d\.\d")` it doesn't work, why?

The reason is that backslashes are “consumed” by a string. Remember, regular strings have their own special characters like `\n`, and a backslash is used for escaping.

Please, take a look, what “`\d\d`” really is:

```
alert("\d\d"); // d.d
```

The quotes “consume” backslashes and interpret them, for instance:

- `\n` – becomes a newline character,
- `\u1234` – becomes the Unicode character with such code,
- ...And when there’s no special meaning: like `\d` or `\z`, then the backslash is simply removed.

So the call to `new RegExp` gets a string without backslashes. That’s why the search doesn’t work!

To fix it, we need to double backslashes, because quotes turn `\\` into `\`:

```
let regStr = "\\d\\.\\d";
alert(regStr); // \d\.\d (correct now)

let reg = new RegExp(regStr);

alert( "Chapter 5.1".match(reg) ); // 5.1
```

Summary

- To search special characters `[\ ^ $. | ? * + ()` literally, we need to prepend them with `\` (“escape them”).
- We also need to escape `/` if we’re inside `/.../` (but not inside `new RegExp`).
- When passing a string `new RegExp`, we need to double backslashes `\\`, cause strings consume one of them.

Sets and ranges [...]

Several characters or character classes inside square brackets `[...]` mean to “search for any character among given”.

Sets

For instance, `[eao]` means any of the 3 characters: 'a', 'e', or 'o'.

That's called a *set*. Sets can be used in a regexp along with regular characters:

```
// find [t or m], and then "op"
alert( "Mop top".match(/[tm]op/gi) ); // "Mop", "top"
```

Please note that although there are multiple characters in the set, they correspond to exactly one character in the match.

So the example below gives no matches:

```
// find "v", then [o or i], then "la"
alert( "Voila".match(/V[oi]la/) ); // null, no matches
```

The pattern assumes:

- `V`,
- then *one* of the letters `[oi]`,
- then `la`.

So there would be a match for `Voila` or `Vila`.

Ranges

Square brackets may also contain *character ranges*.

For instance, `[a-z]` is a character in range from `a` to `z`, and `[0-5]` is a digit from `0` to `5`.

In the example below we're searching for `"x"` followed by two digits or letters from `A` to `F`:

```
alert( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) ); // xAF
```

Please note that in the word `Exception` there's a substring `xce`. It didn't match the pattern, because the letters are lowercase, while in the set `[0-9A-F]` they are uppercase.

If we want to find it too, then we can add a range `a-f`: `[0-9A-Fa-f]`. The `i` flag would allow lowercase too.

Character classes are shorthands for certain character sets.

For instance:

- `\d` – is the same as `[0-9]`,
- `\w` – is the same as `[a-zA-Z0-9_]`,
- `\s` – is the same as `[\t\n\v\f\r]` plus few other unicode space characters.

We can use character classes inside `[...]` as well.

For instance, we want to match all wordly characters or a dash, for words like “twenty-third”. We can’t do it with `\w+`, because `\w` class does not include a dash. But we can use `[\w-]`.

We also can use several classes, for example `[\s\S]` matches spaces or non-spaces – any character. That’s wider than a dot `.`, because the dot matches any character except a newline (unless `s` flag is set).

Excluding ranges

Besides normal ranges, there are “excluding” ranges that look like `[^...]`.

They are denoted by a caret character `^` at the start and match any character *except the given ones*.

For instance:

- `[^aeyo]` – any character except `'a'`, `'e'`, `'y'` or `'o'`.
- `[^0-9]` – any character except a digit, the same as `\D`.
- `[^\s]` – any non-space character, same as `\S`.

The example below looks for any characters except letters, digits and spaces:

```
alert( "alice15@gmail.com".match(/^[^\d\sA-Z]/gi) ); // @ and .
```

No escaping in [...]

Usually when we want to find exactly the dot character, we need to escape it like `\.`. And if we need a backslash, then we use `\\`.

In square brackets the vast majority of special characters can be used without escaping:

- A dot `'.'`.
- A plus `'+'`.
- Parentheses `'()'`.
- Dash `'-'` in the beginning or the end (where it does not define a range).

- A caret `'^'` if not in the beginning (where it means exclusion).
- And the opening square bracket `'['`.

In other words, all special characters are allowed except where they mean something for square brackets.

A dot `"."` inside square brackets means just a dot. The pattern `[.,]` would look for one of characters: either a dot or a comma.

In the example below the regexp `[-().^+]` looks for one of the characters `-` `()` `.` `^` `+`:

```
// No need to escape
let reg = /[-().^+]/g;

alert( "1 + 2 - 3".match(reg) ); // Matches +, -
```

...But if you decide to escape them “just in case”, then there would be no harm:

```
// Escaped everything
let reg = /[\-\(\)\.\^\+]/g;

alert( "1 + 2 - 3".match(reg) ); // also works: +, -
```

✓ Tasks

Java^[^script]

We have a regexp `/Java[^script]/`.

Does it match anything in the string `Java`? In the string `JavaScript`?

[To solution](#)

Find the time as hh:mm or hh-mm

The time can be in the format `hours:minutes` or `hours-minutes`. Both hours and minutes have 2 digits: `09:00` or `21-30`.

Write a regexp to find time:

```
let reg = /your regexp/g;
alert( "Breakfast at 09:00. Dinner at 21-30".match(reg) ); // 09:00, 21-30
```


P.S. In this task we assume that the time is always correct, there's no need to filter out bad strings like "45:67". Later we'll deal with that too.

[To solution](#)

Quantifiers +, *, ? and {n}

Let's say we have a string like `+7(903)-123-45-67` and want to find all numbers in it. But unlike before, we are interested not in single digits, but full numbers: `7`, `903`, `123`, `45`, `67`.

A number is a sequence of 1 or more digits `\d`. To mark how many we need, we need to append a *quantifier*.

Quantity {n}

The simplest quantifier is a number in curly braces: `{n}`.

A quantifier is appended to a character (or a character class, or a `[...]` set etc) and specifies how many we need.

It has a few advanced forms, let's see examples:

The exact count: {5}

`\d{5}` denotes exactly 5 digits, the same as `\d\d\d\d\d`.

The example below looks for a 5-digit number:

```
alert( "I'm 12345 years old".match(/\d{5}/) ); // "12345"
```

We can add `\b` to exclude longer numbers: `\b\d{5}\b`.

The range: {3, 5}, match 3-5 times

To find numbers from 3 to 5 digits we can put the limits into curly braces: `\d{3,5}`

```
alert( "I'm not 12, but 1234 years old".match(/\d{3,5}/) ); // "1234"
```

We can omit the upper limit.

Then a regexp `\d{3,}` looks for sequences of digits of length 3 or more:

```
alert( "I'm not 12, but 345678 years old".match(/\\d{3,}/) ); // "345678"
```

Let's return to the string `+7(903)-123-45-67`.

A number is a sequence of one or more digits in a row. So the regexp is `\\d{1,}`:

```
let str = "+7(903)-123-45-67";  
  
let numbers = str.match(/\\d{1,}/g);  
  
alert(numbers); // 7,903,123,45,67
```

Shorthands

There are shorthands for most used quantifiers:

+

Means “one or more”, the same as `{1,}`.

For instance, `\\d+` looks for numbers:

```
let str = "+7(903)-123-45-67";  
  
alert( str.match(/\\d+/g) ); // 7,903,123,45,67
```

?

Means “zero or one”, the same as `{0,1}`. In other words, it makes the symbol optional.

For instance, the pattern `ou?r` looks for `o` followed by zero or one `u`, and then `r`.

So, `colou?r` finds both `color` and `colour`:

```
let str = "Should I write color or colour?";  
  
alert( str.match(/colou?r/g) ); // color, colour
```

Means “zero or more”, the same as `{0,}`. That is, the character may repeat any times or be absent.

For example, `\\d0*` looks for a digit followed by any number of zeroes:

```
alert( "100 10 1".match(/d0*/g) ); // 100, 10, 1
```

Compare it with '+' (one or more):

```
alert( "100 10 1".match(/d0+/g) ); // 100, 10  
// 1 not matched, as 0+ requires at least one zero
```

More examples

Quantifiers are used very often. They serve as the main “building block” of complex regular expressions, so let's see more examples.

Regex “decimal fraction” (a number with a floating point): `\d+\.\d+`

In action:

```
alert( "0 1 12.345 7890".match(/d+\.\d+/g) ); // 12.345
```

Regex “open HTML-tag without attributes”, like `` or `<p>`: `/<[a-z]+>/i`

In action:

```
alert( "<body> ... </body>".match(/<[a-z]+>/gi) ); // <body>
```

We look for character `'<'` followed by one or more Latin letters, and then `'>'`.

Regex “open HTML-tag without attributes” (improved): `/<[a-z][a-z0-9]*>/i`

Better regexp: according to the standard, HTML tag name may have a digit at any position except the first one, like `<h1>`.

```
alert( "<h1>Hi!</h1>".match(/<[a-z][a-z0-9]*>/gi) ); // <h1>
```

Regex “opening or closing HTML-tag without attributes”: `/<\/?[a-z][a-z0-9]*>/i`

We added an optional slash `/?` before the tag. Had to escape it with a backslash, otherwise JavaScript would think it is the pattern end.

```
alert( "<h1>Hi!</h1>".match(/<\/?[a-z][a-z0-9]*>/gi) ); // <h1>, </h1>
```

i To make a regexp more precise, we often need make it more complex

We can see one common rule in these examples: the more precise is the regular expression – the longer and more complex it is.

For instance, for HTML tags we could use a simpler regexp: <\w+>.

...But because \w means any Latin letter or a digit or '_', the regexp also matches non-tags, for instance <_>. So it's much simpler than <[a-z][a-z0-9]*>, but less reliable.

Are we ok with <\w+> or we need <[a-z][a-z0-9]*>?

In real life both variants are acceptable. Depends on how tolerant we can be to “extra” matches and whether it's difficult or not to filter them out by other means.

✓ Tasks

How to find an ellipsis "..." ?

importance: 5

Create a regexp to find ellipsis: 3 (or more?) dots in a row.

Check it:

```
let reg = /your regexp/g;  
alert( "Hello!... How goes?.....".match(reg) ); // ..., .....
```

[To solution](#)

Regexp for HTML colors

Create a regexp to search HTML-colors written as #ABCDEF: first # and then 6 hexadimal characters.

An example of use:

```
let reg = /...your regexp.../  
  
let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2 #1234567890";  
  
alert( str.match(reg) ) // #121212,#AA00ef
```

P.S. In this task we do not need other color formats like `#123` or `rgb(1, 2, 3)` etc.

[To solution](#)

Greedy and lazy quantifiers

Quantifiers are very simple from the first sight, but in fact they can be tricky.

We should understand how the search works very well if we plan to look for something more complex than `/\d+/`.

Let's take the following task as an example.

We have a text and need to replace all quotes `"..."` with guillemet marks: `«...»`. They are preferred for typography in many countries.

For instance: `"Hello, world"` should become `«Hello, world»`. Some countries prefer other quotes, like `„Witam, świat!”` (Polish) or `「你好, 世界」` (Chinese), but for our task let's choose `«...»`.

The first thing to do is to locate quoted strings, and then we can replace them.

A regular expression like `/".+"/g` (a quote, then something, then the other quote) may seem like a good fit, but it isn't!

Let's try it:

```
let reg = /".+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(reg) ); // "witch" and her "broom"
```

...We can see that it works not as intended!

Instead of finding two matches `"witch"` and `"broom"`, it finds one: `"witch" and her "broom"`.

That can be described as “greediness is the cause of all evil”.

Greedy search

To find a match, the regular expression engine uses the following algorithm:

- For every position in the string
 - Match the pattern at that position.

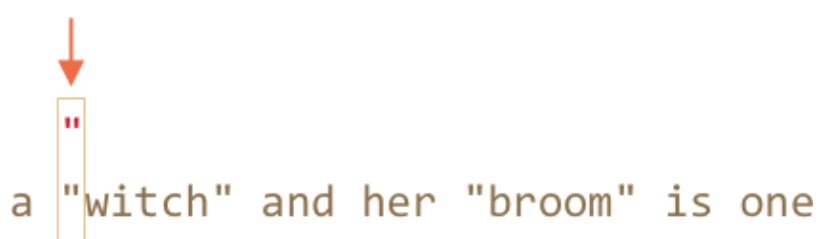
- If there's no match, go to the next position.

These common words do not make it obvious why the regexp fails, so let's elaborate how the search works for the pattern `" .+ "`.

1. The first pattern character is a quote `"`.

The regular expression engine tries to find it at the zero position of the source string `a "witch" and her "broom" is one`, but there's `a` there, so there's immediately no match.

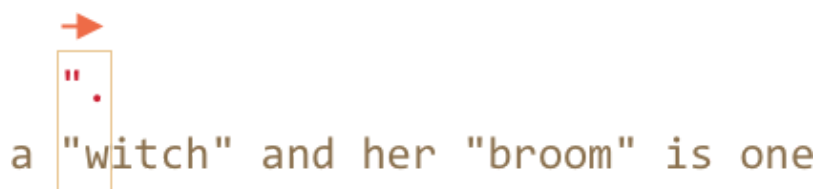
Then it advances: goes to the next positions in the source string and tries to find the first character of the pattern there, and finally finds the quote at the 3rd position:



a "witch" and her "broom" is one

2. The quote is detected, and then the engine tries to find a match for the rest of the pattern. It tries to see if the rest of the subject string conforms to `.+ "`.

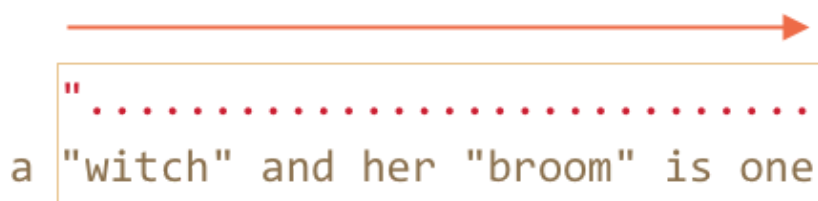
In our case the next pattern character is `.` (a dot). It denotes "any character except a newline", so the next string letter `'w'` fits:



a "witch" and her "broom" is one

3. Then the dot repeats because of the quantifier `.+` . The regular expression engine builds the match by taking characters one by one while it is possible.

...When does it become impossible? All characters match the dot, so it only stops when it reaches the end of the string:

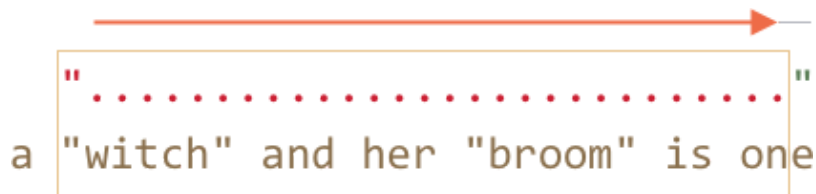


a "witch" and her "broom" is one

4. Now the engine finished repeating for `.+` and tries to find the next character of the pattern. It's the quote `"`. But there's a problem: the string has finished, there are no more characters!

The regular expression engine understands that it took too many `.+` and starts to *backtrack*.

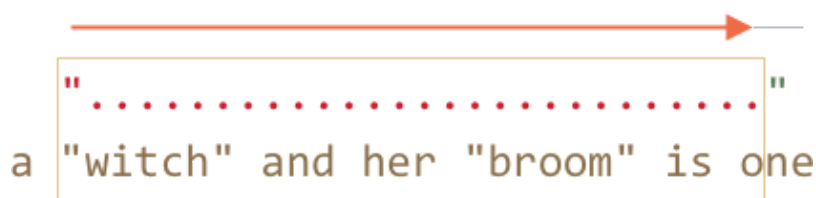
In other words, it shortens the match for the quantifier by one character:



Now it assumes that `.+` ends one character before the end and tries to match the rest of the pattern from that position.

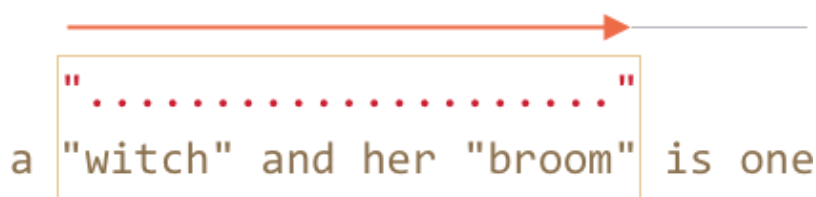
If there were a quote there, then that would be the end, but the last character is `'e'`, so there's no match.

5. ...So the engine decreases the number of repetitions of `.+` by one more character:



The quote `'\"'` does not match `'n'`.

6. The engine keep backtracking: it decreases the count of repetition for `'.'` until the rest of the pattern (in our case `'\"'`) matches:



7. The match is complete.
8. So the first match is `"witch" and her "broom"`. The further search starts where the first match ends, but there are no more quotes in the rest of the string `is one`, so no more results.

That's probably not what we expected, but that's how it works.

In the greedy mode (by default) the quantifier is repeated as many times as possible.

The regexp engine tries to fetch as many characters as it can by `.+`, and then shortens that one by one.

For our task we want another thing. That's what the lazy quantifier mode is for.

Lazy mode

The lazy mode of quantifier is an opposite to the greedy mode. It means: “repeat minimal number of times”.

We can enable it by putting a question mark `'?'` after the quantifier, so that it becomes `*?` or `+?` or even `??` for `'?'`.

To make things clear: usually a question mark `?` is a quantifier by itself (zero or one), but if added *after another quantifier (or even itself)* it gets another meaning – it switches the matching mode from greedy to lazy.

The regexp `/".+?"/g` works as intended: it finds `"witch"` and `"broom"`:

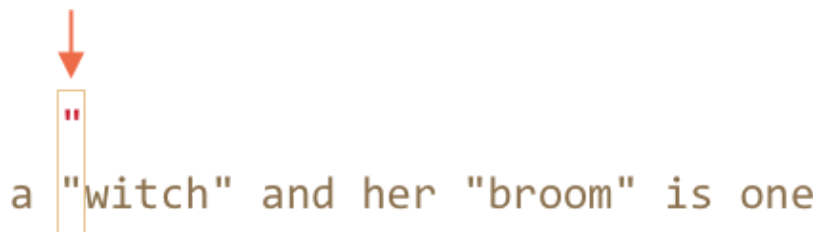
```
let reg = /".+?"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(reg) ); // witch, broom
```

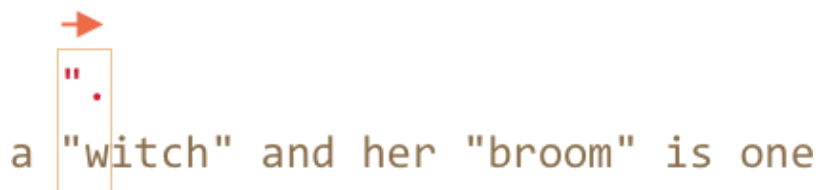
To clearly understand the change, let's trace the search step by step.

1. The first step is the same: it finds the pattern start `'"'` at the 3rd position:



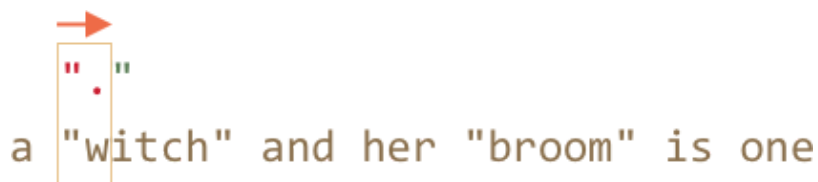
a "witch" and her "broom" is one

2. The next step is also similar: the engine finds a match for the dot `'.'`:



a "witch" and her "broom" is one

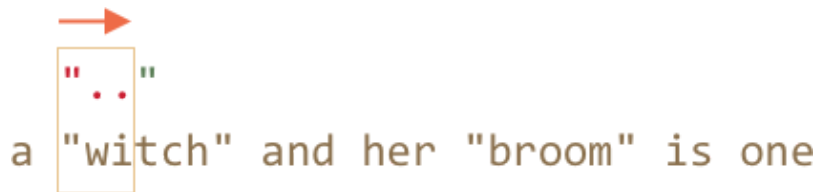
3. And now the search goes differently. Because we have a lazy mode for `+?`, the engine doesn't try to match a dot one more time, but stops and tries to match the rest of the pattern `'"'` right now:



a "witch" and her "broom" is one

If there were a quote there, then the search would end, but there's 'i', so there's no match.

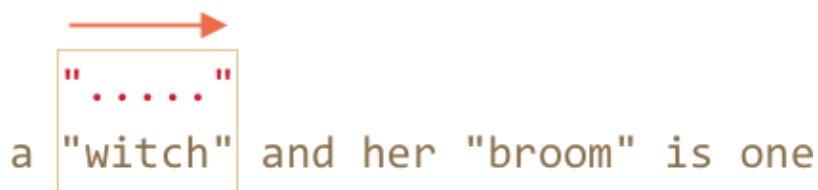
4. Then the regular expression engine increases the number of repetitions for the dot and tries one more time:



a "witch" and her "broom" is one

Failure again. Then the number of repetitions is increased again and again...

5. ...Till the match for the rest of the pattern is found:



a "witch" and her "broom" is one

6. The next search starts from the end of the current match and yield one more result:



a "witch" and her "broom" is one

In this example we saw how the lazy mode works for +?. Quantifiers +? and ?? work the similar way – the regexp engine increases the number of repetitions only if the rest of the pattern can't match on the given position.

Laziness is only enabled for the quantifier with ?.

Other quantifiers remain greedy.

For instance:

```
alert( "123 456".match(/\\d+ \\d+?/g) ); // 123 4
```

1. The pattern \\d+ tries to match as many numbers as it can (greedy mode), so it finds 123 and stops, because the next character is a space ' '.
2. Then there's a space in pattern, it matches.
3. Then there's \\d+?. The quantifier is in lazy mode, so it finds one digit 4 and tries to check if the rest of the pattern matches from there.

...But there's nothing in the pattern after `\d+?`.

The lazy mode doesn't repeat anything without a need. The pattern finished, so we're done. We have a match `123 4`.

4. The next search starts from the character `5`.

Optimizations

Modern regular expression engines can optimize internal algorithms to work faster. So they may work a bit different from the described algorithm.

But to understand how regular expressions work and to build regular expressions, we don't need to know about that. They are only used internally to optimize things.

Complex regular expressions are hard to optimize, so the search may work exactly as described as well.

Alternative approach

With regexps, there's often more than one way to do the same thing.

In our case we can find quoted strings without lazy mode using the regexp `"^[^"]+"`:

```
let reg = /^[^"]+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(reg) ); // witch, broom
```

The regexp `"^[^"]+"` gives correct results, because it looks for a quote `'"` followed by one or more non-quotes `^[^"]`, and then the closing quote.

When the regexp engine looks for `^[^"]+` it stops the repetitions when it meets the closing quote, and we're done.

Please note, that this logic does not replace lazy quantifiers!

It is just different. There are times when we need one or another.

Let's see an example where lazy quantifiers fail and this variant works right.

For instance, we want to find links of the form ``, with any href.

Which regular expression to use?

The first idea might be: `//g`.

Let's check it:

```
let str = '...<a href="link" class="doc">...';
let reg = /<a href=".*" class="doc">/g;

// Works!
alert( str.match(reg) ); // <a href="link" class="doc">
```

It worked. But let's see what happens if there are many links in the text?

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let reg = /<a href=".*" class="doc">/g;

// Whoops! Two links in one match!
alert( str.match(reg) ); // <a href="link1" class="doc">... <a href="link2" class="doc">
```

Now the result is wrong for the same reason as our “witches” example. The quantifier `.*` took too many characters.

The match looks like this:

```
<a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

Let's modify the pattern by making the quantifier `.*?` lazy:

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let reg = /<a href=".*?" class="doc">/g;

// Works!
alert( str.match(reg) ); // <a href="link1" class="doc">, <a href="link2" class="doc">
```

Now it seems to work, there are two matches:

```
<a href="....." class="doc">    <a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

...But let's test it on one more text input:

```
let str = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let reg = /<a href=".*?" class="doc">/g;

// Wrong match!
alert( str.match(reg) ); // <a href="link1" class="wrong">... <p style="" class="doc">...
```

Now it fails. The match includes not just a link, but also a lot of text after it, including `<p...>`.

Why?

That's what's going on:

1. First the regexp finds a link start `<a href="`.
2. Then it looks for `. *?`: takes one character (lazily!), check if there's a match for `" class="doc">` (none).
3. Then takes another character into `. *?`, and so on... until it finally reaches `" class="doc">`.

But the problem is: that's already beyond the link, in another tag `<p>`. Not what we want.

Here's the picture of the match aligned with the text:

```
<a href="....." class="doc">
<a href="link1" class="wrong">... <p style="" class="doc">
```

So the laziness did not work for us here.

We need the pattern to look for ``, but both greedy and lazy variants have problems.

The correct variant would be: `href="[^] *"`. It will take all characters inside the `href` attribute till the nearest quote, just what we need.

A working example:

```
let str1 = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let str2 = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let reg = /<a href="[ ^ ] *" class="doc">/g;

// Works!
alert( str1.match(reg) ); // null, no matches, that's correct
alert( str2.match(reg) ); // <a href="link1" class="doc">, <a href="link2" class="doc">
```

Summary

Quantifiers have two modes of work:

Greedy

By default the regular expression engine tries to repeat the quantifier as many times as possible. For instance, `\d+` consumes all possible digits. When it becomes impossible to consume more (no more digits or string end), then it continues to match the rest of the pattern. If there's no match then it decreases the number of repetitions (backtracks) and tries again.

Lazy

Enabled by the question mark `?` after the quantifier. The regexp engine tries to match the rest of the pattern before each repetition of the quantifier.

As we've seen, the lazy mode is not a "panacea" from the greedy search. An alternative is a "fine-tuned" greedy search, with exclusions. Soon we'll see more examples of it.

✓ Tasks

A match for `/d+? d+?/`

What's the match here?

```
"123 456".match(/d+? d+?/g) ); // ?
```

[To solution](#)

Find HTML comments

Find all HTML comments in the text:

```
let reg = /your regexp/g;

let str = `... <!-- My -- comment
test --> .. <!--> ..
`;

alert( str.match(reg) ); // '<!-- My -- comment \n test -->', '<!-->'
```

[To solution](#)

Find HTML tags

Create a regular expression to find all (opening and closing) HTML tags with their attributes.

An example of use:

```
let reg = /your regexp/g;

let str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(reg) ); // '<a href="/">', '<input type="radio" checked>', '<b>'
```

Here we assume that tag attributes may not contain `<` and `>` (inside quotes too), that simplifies things a bit.

[To solution](#)

Capturing groups

A part of a pattern can be enclosed in parentheses `(. . .)`. This is called a “capturing group”.

That has two effects:

1. It allows to place a part of the match into a separate array.
2. If we put a quantifier after the parentheses, it applies to the parentheses as a whole, not the last character.

Example

In the example below the pattern `(go)+` finds one or more `'go'`:

```
alert( 'Gogogo now!'.match(/(go)+/i) ); // "Gogogo"
```

Without parentheses, the pattern `/go+/?` means `g`, followed by `o` repeated one or more times. For instance, `goooo` or `goooooooooo`.

Parentheses group the word `(go)` together.

Let's make something more complex – a regexp to match an email.

Examples of emails:

```
my@mail.com
john.smith@site.com.uk
```

The pattern: `[- . \w] + @ ([\w -] + \.) + [\w -] { 2 , 20 } .`

1. The first part `[- . \w] +` (before `@`) may include any alphanumeric word characters, a dot and a dash, to match `john.smith`.
2. Then `@`, and the domain. It may be a subdomain like `host.site.com.uk`, so we match it as "a word followed by a dot `([\w -] + \.)` (repeated), and then the last part must be a word: `com` or `uk` (but not very long: 2-20 characters).

That regexp is not perfect, but good enough to fix errors or occasional mistypes.

For instance, we can find all emails in the string:

```
let reg = /[ - . \w ] + @ ( [ \w - ] + \. ) + [ \w - ] { 2 , 20 } /g;

alert("my@mail.com @ his@site.com.uk".match(reg)); // my@mail.com, his@site.com.uk
```

In this example parentheses were used to make a group for repeating `(. . .) +`. But there are other uses too, let's see them.

Contents of parentheses

Parentheses are numbered from left to right. The search engine remembers the content matched by each of them and allows to reference it in the pattern or in the replacement string.

For instance, we'd like to find HTML tags `< . * ? >`, and process them.

Let's wrap the inner content into parentheses, like this: `< (. * ?) >`.

We'll get both the tag as a whole and its content as an array:

```
let str = '<h1>Hello, world!</h1>';
let reg = /<(.*?)>/;

alert( str.match(reg) ); // Array: ["<h1>", "h1"]
```

The call to [String#match](#) [↗](#) returns groups only if the regexp only looks for the first match, that is: has no `/ . . . /g` flag.

If we need all matches with their groups then we can use `.matchAll` or `regexp.exec` as described in [Methods of RegExp and String](#):

```
let str = '<h1>Hello, world!</h1>';

// two matches: opening <h1> and closing </h1> tags
let reg = /<(.*?)>/g;

let matches = Array.from( str.matchAll(reg) );

alert(matches[0]); // Array: ["<h1>", "h1"]
alert(matches[1]); // Array: ["</h1>", "/h1"]
```

Here we have two matches for <(.*?)>, each of them is an array with the full match and groups.

Nested groups

Parentheses can be nested. In this case the numbering also goes from left to right.

For instance, when searching a tag in we may be interested in:

1. The tag content as a whole: span class="my".
2. The tag name: span.
3. The tag attributes: class="my".

Let's add parentheses for them:

```
let str = '<span class="my">';

let reg = /<(([a-z]+)\s*([>]*))>/;

let result = str.match(reg);
alert(result); // <span class="my">, span class="my", span, class="my"
```

Here's how groups look:

```

      span class="my"
    1 |-----|
    <(([a-z]+)\s*([>]*))>
      2 |-----| 3 |-----|
      span      class="my"
```

At the zero index of the `result` is always the full match.

Then groups, numbered from left to right. Whichever opens first gives the first group `result[1]`. Here it encloses the whole tag content.

Then in `result[2]` goes the group from the second opening `(` till the corresponding `)` – tag name, then we don't group spaces, but group attributes for `result[3]`.

If a group is optional and doesn't exist in the match, the corresponding `result` index is present (and equals `undefined`).

For instance, let's consider the regexp `a(z)?(c)?`. It looks for `"a"` optionally followed by `"z"` optionally followed by `"c"`.

If we run it on the string with a single letter `a`, then the result is:

```
let match = 'a'.match(/a(z)?(c)?/);

alert( match.length ); // 3
alert( match[0] ); // a (whole match)
alert( match[1] ); // undefined
alert( match[2] ); // undefined
```

The array has the length of `3`, but all groups are empty.

And here's a more complex match for the string `ack`:

```
let match = 'ack'.match(/a(z)?(c)?/)

alert( match.length ); // 3
alert( match[0] ); // ac (whole match)
alert( match[1] ); // undefined, because there's nothing for (z)?
alert( match[2] ); // c
```

The array length is permanent: `3`. But there's nothing for the group `(z)?`, so the result is `["ac", undefined, "c"]`.

Named groups

Remembering groups by their numbers is hard. For simple patterns it's doable, but for more complex ones we can give names to parentheses.

That's done by putting `?<name>` immediately after the opening paren, like this:

```
let dateRegExp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;
let str = "2019-04-30";

let groups = str.match(dateRegExp).groups;

alert(groups.year); // 2019
```

```
alert(groups.month); // 04
alert(groups.day); // 30
```

As you can see, the groups reside in the `.groups` property of the match.

We can also use them in the replacement string, as `$<name>` (like `$1` . . . `9` , but a name instead of a digit).

For instance, let's reformat the date into `day.month.year` :

```
let dateRegex = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;

let str = "2019-04-30";

let rearranged = str.replace(dateRegex, '$<day>.$<month>.$<year>');

alert(rearranged); // 30.04.2019
```

If we use a function for the replacement, then named `groups` object is always the last argument:

```
let dateRegex = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/;

let str = "2019-04-30";

let rearranged = str.replace(dateRegex,
  (str, year, month, day, offset, input, groups) =>
    `${groups.day}.${groups.month}.${groups.year}`
);

alert(rearranged); // 30.04.2019
```

Usually, when we intend to use named groups, we don't need positional arguments of the function. For the majority of real-life cases we only need `str` and `groups` .

So we can write it a little bit shorter:

```
let rearranged = str.replace(dateRegex, (str, ...args) => {
  let {year, month, day} = args.pop();
  alert(str); // 2019-04-30
  alert(year); // 2019
  alert(month); // 04
  alert(day); // 30
});
```

Non-capturing groups with ?:

Sometimes we need parentheses to correctly apply a quantifier, but we don't want the contents in results.

A group may be excluded by adding `?:` in the beginning.

For instance, if we want to find `(go)+`, but don't want to remember the contents `(go)` in a separate array item, we can write: `(?:go)+`.

In the example below we only get the name "John" as a separate member of the `results` array:

```
let str = "Gogo John!";
// exclude Gogo from capturing
let reg = /(?:go)+ (\w+)/i;

let result = str.match(reg);

alert( result.length ); // 2
alert( result[1] ); // John
```

Summary

Parentheses group together a part of the regular expression, so that the quantifier applies to it as a whole.

Parentheses groups are numbered left-to-right, and can optionally be named with `(?<name>...)`.

The content, matched by a group, can be referenced both in the replacement string as `$1`, `$2` etc, or by the name `$name` if named.

So, parentheses groups are called "capturing groups", as they "capture" a part of the match. We get that part separately from the result as a member of the array or in `.groups` if it's named.

We can exclude the group from remembering (make it "non-capturing") by putting `?:` at the start: `(?:...)`, that's used if we'd like to apply a quantifier to the whole group, but don't need it in the result.

✓ Tasks

Find color in the format `#abc` or `#abcdef`

Write a RegExp that matches colors in the format `#abc` or `#abcdef`. That is: `#` followed by 3 or 6 hexadecimal digits.

Usage example:

```
let reg = /your regexp/g;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(reg) ); // #3f3 #AA00ef
```

P.S. This should be exactly 3 or 6 hex digits: values like `#abcd` should not match.

[To solution](#)

Find positive numbers

Create a regexp that looks for positive numbers, including those without a decimal point.

An example of use:

```
let reg = /your regexp/g;

let str = "1.5 0 -5 12. 123.4.";

alert( str.match(reg) ); // 1.5, 12, 123.4 (ignores 0 and -5)
```

[To solution](#)

Find all numbers

Write a regexp that looks for all decimal numbers including integer ones, with the floating point and negative ones.

An example of use:

```
let reg = /your regexp/g;

let str = "-1.5 0 2 -123.4.";

alert( str.match(re) ); // -1.5, 0, 2, -123.4
```

[To solution](#)

Parse an expression

An arithmetical expression consists of 2 numbers and an operator between them, for instance:

- `1 + 2`
- `1.2 * 3.4`
- `-3 / -6`
- `-2 - 2`

The operator is one of: `"+"`, `"-"`, `"*"` or `"/"`.

There may be extra spaces at the beginning, at the end or between the parts.

Create a function `parse(expr)` that takes an expression and returns an array of 3 items:

1. The first number.
2. The operator.
3. The second number.

For example:

```
let [a, op, b] = parse("1.2 * 3.4");

alert(a); // 1.2
alert(op); // *
alert(b); // 3.4
```

[To solution](#)

Backreferences in pattern: `\n` and `\k`

We can use the contents of capturing groups `(...)` not only in the result or in the replacement string, but also in the pattern itself.

Backreference by number: `\n`

A group can be referenced in the pattern using `\n`, where `n` is the group number.

To make things clear let's consider a task.

We need to find a quoted string: either a single-quoted `'...'` or a double-quoted `"..."` – both variants need to match.

How to look for them?

We can put both kinds of quotes in the square brackets: `['"] (.*?) ['"]`, but it would find strings with mixed quotes, like `"...'` and `'..."`. That would lead to incorrect matches when one quote appears inside other ones, like the string `"She's the one!"`:

```
let str = `He said: "She's the one!".`;

let reg = /[ '"] ( .*? ) [ '"] /g;

// The result is not what we expect
alert( str.match(reg) ); // "She'
```

As we can see, the pattern found an opening quote `"`, then the text is consumed lazily till the other quote `'`, that closes the match.

To make sure that the pattern looks for the closing quote exactly the same as the opening one, we can wrap it into a capturing group and use the backreference.

Here's the correct code:

```
let str = `He said: "She's the one!".`;

let reg = /([ '"] ) ( .*? ) \1 /g;

alert( str.match(reg) ); // "She's the one!"
```

Now it works! The regular expression engine finds the first quote `(['"])` and remembers the content of `(. . .)`, that's the first capturing group.

Further in the pattern `\1` means “find the same text as in the first group”, exactly the same quote in our case.

Please note:

- To reference a group inside a replacement string – we use `$1`, while in the pattern – a backslash `\1`.
- If we use `?:` in the group, then we can't reference it. Groups that are excluded from capturing `(?: . . .)` are not remembered by the engine.

Backreference by name: `\k<name>`

For named groups, we can backreference by `\k<name>`.

The same example with the named group:

```
let str = `He said: "She's the one!".`;

let reg = /(?!<quote>['"])(.*?)\k<quote>/g;

alert( str.match(reg) ); // "She's the one!"
```

Alternation (OR) |

Alternation is the term in regular expression that is actually a simple “OR”.

In a regular expression it is denoted with a vertical line character |.

For instance, we need to find programming languages: HTML, PHP, Java or JavaScript.

The corresponding regexp: html|php|java(script)?.

A usage example:

```
let reg = /html|php|css|java(script)?/gi;

let str = "First HTML appeared, then CSS, then JavaScript";

alert( str.match(reg) ); // 'HTML', 'CSS', 'JavaScript'
```

We already know a similar thing – square brackets. They allow to choose between multiple character, for instance gr[ae]y matches gray or grey.

Square brackets allow only characters or character sets. Alternation allows any expressions. A regexp A|B|C means one of expressions A, B or C.

For instance:

- gr(a|e)y means exactly the same as gr[ae]y.
- gr|ey means gr or ey.

To separate a part of the pattern for alternation we usually enclose it in parentheses, like this: before(XXX|YYY)after.

Regex for time

In previous chapters there was a task to build a regexp for searching time in the form hh:mm, for instance 12:00. But a simple \d\d:\d\d is too vague. It accepts 25:99 as the time (as 99 seconds match the pattern).

How can we make a better one?

We can apply more careful matching. First, the hours:

- If the first digit is `0` or `1`, then the next digit can be anything.
- Or, if the first digit is `2`, then the next must be `[0-3]`.

As a regexp: `[01]\d|2[0-3]`.

Next, the minutes must be from `0` to `59`. In the regexp language that means `[0-5]\d`: the first digit `0-5`, and then any digit.

Let's glue them together into the pattern: `[01]\d|2[0-3]:[0-5]\d`.

We're almost done, but there's a problem. The alternation `|` now happens to be between `[01]\d` and `2[0-3]:[0-5]\d`.

That's wrong, as it should be applied only to hours `[01]\d` OR `2[0-3]`. That's a common mistake when starting to work with regular expressions.

The correct variant:

```
let reg = /([01]\d|2[0-3]):[0-5]\d/g;

alert("00:00 10:10 23:59 25:99 1:2".match(reg)); // 00:00,10:10,23:59
```

✓ Tasks

Find programming languages

There are many programming languages, for instance Java, JavaScript, PHP, C, C++.

Create a regexp that finds them in the string `Java JavaScript PHP C++ C`:

```
let reg = /your regexp/g;

alert("Java JavaScript PHP C++ C".match(reg)); // Java JavaScript PHP C++ C
```

To solution

Find bbtag pairs

A “bb-tag” looks like `[tag]...[/tag]`, where `tag` is one of: `b`, `url` or `quote`.

For instance:


```
[b]text[/b]
[url]http://google.com[/url]
```

BB-tags can be nested. But a tag can't be nested into itself, for instance:

```
Normal:
[url] [b]http://google.com[/b] [/url]
[quote] [b]text[/b] [/quote]

Impossible:
[b][b]text[/b][/b]
```

Tags can contain line breaks, that's normal:

```
[quote]
  [b]text[/b]
[/quote]
```

Create a regexp to find all BB-tags with their contents.

For instance:

```
let reg = /your regexp/flags;

let str = "..[url]http://google.com[/url]..";
alert( str.match(reg) ); // [url]http://google.com[/url]
```

If tags are nested, then we need the outer tag (if we want we can continue the search in its content):

```
let reg = /your regexp/flags;

let str = "..[url][b]http://google.com[/b][/url]..";
alert( str.match(reg) ); // [url][b]http://google.com[/b][/url]
```

To solution

Find quoted strings

Create a regexp to find strings in double quotes "...".

The strings should support escaping, the same way as JavaScript strings do. For instance, quotes can be inserted as \" a newline as \n, and the slash itself as \\.

```
let str = "Just like \"here\".";
```

Please note, in particular, that an escaped quote `\"` does not end a string.

So we should search from one quote to the other ignoring escaped quotes on the way.

That's the essential part of the task, otherwise it would be trivial.

Examples of strings to match:

```
.. "test me" ..  
.. "Say \"Hello\"" ... (escaped quotes inside)  
.. "\"\" .. (double slash inside)  
.. "\"\" \"\" .. (double slash and an escaped quote inside)
```

In JavaScript we need to double the slashes to pass them right into the string, like this:

```
let str = ' .. "test me" .. "Say \\\"Hello\\\"" .. "\\\" \\\"" .. ' ;  
  
// the in-memory string  
alert(str); // .. "test me" .. "Say \"Hello\"" .. "\"\" \"\" ..
```

[To solution](#)

Find the full tag

Write a regexp to find the tag `<style...>`. It should match the full tag: it may have no attributes `<style>` or have several of them `<style type="..." id="...">`.

...But the regexp should not match `<styler>`!

For instance:

```
let reg = /your regexp/g;  
  
alert( '<style> <styler> <style test="...">'.match(reg) ); // <style>, <style test=
```

[To solution](#)

String start ^ and finish \$

The caret `'^'` and dollar `'$'` characters have special meaning in a regexp. They are called “anchors”.

The caret `^` matches at the beginning of the text, and the dollar `$` – in the end.

For instance, let's test if the text starts with `Mary`:

```
let str1 = "Mary had a little lamb, it's fleece was white as snow";
let str2 = 'Everywhere Mary went, the lamp was sure to go';

alert( /^Mary/.test(str1) ); // true
alert( /^Mary/.test(str2) ); // false
```

The pattern `^Mary` means: “the string start and then Mary”.

Now let's test whether the text ends with an email.

To match an email, we can use a regexp `[-.\w]+@([\w-]+\.)+[\w-]{2,20}`.

To test whether the string ends with the email, let's add `$` to the pattern:

```
let reg = /[-.\w]+@([\w-]+\.)+[\w-]{2,20}$/g;

let str1 = 'My email is mail@site.com';
let str2 = 'Everywhere Mary went, the lamp was sure to go';

alert( reg.test(str1) ); // true
alert( reg.test(str2) ); // false
```

We can use both anchors together to check whether the string exactly follows the pattern. That's often used for validation.

For instance we want to check that `str` is exactly a color in the form `#` plus 6 hex digits. The pattern for the color is `#[0-9a-f]{6}`.

To check that the *whole string* exactly matches it, we add `^...$`:

```
let str = "#abcdef";

alert( /^#[0-9a-f]{6}$/i.test(str) ); // true
```

The regexp engine looks for the text start, then the color, and then immediately the text end. Just what we need.

Anchors have zero length

Anchors just like `\b` are tests. They have zero-width.

In other words, they do not match a character, but rather force the regexp engine to check the condition (text start/end).

The behavior of anchors changes if there's a flag `m` (multiline mode). We'll explore it in the next chapter.

Tasks

Regex `^$`

Which string matches the pattern `^$`?

[To solution](#)

Check MAC-address

[MAC-address](#) of a network interface consists of 6 two-digit hex numbers separated by a colon.

For instance: `'01:32:54:67:89:AB'`.

Write a regexp that checks whether a string is MAC-address.

Usage:

```
let reg = /your regexp/;

alert( reg.test('01:32:54:67:89:AB') ); // true

alert( reg.test('0132546789AB') ); // false (no colons)

alert( reg.test('01:32:54:67:89') ); // false (5 numbers, must be 6)

alert( reg.test('01:32:54:67:89:ZZ') ); // false (ZZ at the end)
```

[To solution](#)

Multiline mode, flag "m"

The multiline mode is enabled by the flag /.../m.

It only affects the behavior of ^ and \$.

In the multiline mode they match not only at the beginning and end of the string, but also at start/end of line.

Line start ^

In the example below the text has multiple lines. The pattern /^\d+/gm takes a number from the beginning of each one:

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert( str.match(/^\d+/gm) ); // 1, 2, 33
```

The regexp engine moves along the text and looks for a line start ^, when finds – continues to match the rest of the pattern \d+.

Without the flag /.../m only the first number is matched:

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert( str.match(/^\d+/g) ); // 1
```

That's because by default a caret ^ only matches at the beginning of the text, and in the multiline mode – at the start of any line.

Line end \$

The dollar sign \$ behaves similarly.

The regular expression \w+\$ finds the last word in every line

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert( str.match(/\w+$/gm) ); // Winnie,Piglet,Eeyore
```

Without the `/.../m` flag the dollar `$` would only match the end of the whole string, so only the very last word would be found.

Anchors `^` `$` versus `\n`

To find a newline, we can use not only `^` and `$`, but also the newline character `\n`.

The first difference is that unlike anchors, the character `\n` “consumes” the newline character and adds it to the result.

For instance, here we use it instead of `$`:

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert( str.match(/\w+\n/gim) ); // Winnie\n,Piglet\n
```

Here every match is a word plus a newline character.

And one more difference – the newline `\n` does not match at the string end. That’s why `Eeyore` is not found in the example above.

So, anchors are usually better, they are closer to what we want to get.

Lookahead and lookbehind

Sometimes we need to match a pattern only if followed by another pattern. For instance, we’d like to get the price from a string like `1 turkey costs 30€`.

We need a number (let’s say a price has no decimal point) followed by `€` sign.

That’s what lookahead is for.

Lookahead

The syntax is: `x(?=y)`, it means “look for `x`, but match only if followed by `y`”.

For an integer amount followed by `€`, the regexp will be `\d+(?=€)`:

```
let str = "1 turkey costs 30€";

alert( str.match(/\d+(?=€)/) ); // 30 (correctly skipped the sole number 1)
```

Let’s say we want a quantity instead, that is a number, NOT followed by `€`.

Here a negative lookahead can be applied.

The syntax is: `x(?!y)`, it means "search `x`, but only if not followed by `y`".

```
let str = "2 turkeys cost 60€";  
  
alert( str.match(/\d+(?!€)/) ); // 2 (correctly skipped the price)
```

Lookbehind

Lookahead allows to add a condition for "what goes after".

Lookbehind is similar, but it looks behind. That is, it allows to match a pattern only if there's something before.

The syntax is:

- Positive lookbehind: `(?<=y)x`, matches `x`, but only if it follows after `y`.
- Negative lookbehind: `(?<!\y)x`, matches `x`, but only if there's no `y` before.

For example, let's change the price to US dollars. The dollar sign is usually before the number, so to look for `$30` we'll use `(?<=\$)\d+` – an amount preceded by `$`:

```
let str = "1 turkey costs $30";  
  
alert( str.match(/(?<=\$)\d+/) ); // 30 (skipped the sole number)
```

And, to find the quantity – a number, not preceded by `$`, we can use a negative lookbehind `(?<!\$)\d+`:

```
let str = "2 turkeys cost $60";  
  
alert( str.match(/(?<!\$)\d+/) ); // 2 (skipped the price)
```

Capture groups

Generally, what's inside the lookahead (a common name for both lookahead and lookbehind) parentheses does not become a part of the match.

E.g. in the pattern `\d+(?=€)`, the `€` sign doesn't get captured as a part of the match. That's natural: we look for a number `\d+`, while `(?=€)` is just a test that it should be followed by `€`.

But in some situations we might want to capture the lookahead expression as well, or a part of it. That's possible. Just wrap that into additional parentheses.

For instance, here the currency (€|kr) is captured, along with the amount:

```
let str = "1 turkey costs 30€";
let reg = /\d+(?=(€|kr))/; // extra parentheses around €|kr

alert( str.match(reg) ); // 30, €
```

And here's the same for lookbehind:

```
let str = "1 turkey costs $30";
let reg = /(?<=(\$/£))\d+/;

alert( str.match(reg) ); // 30, $
```

Please note that for lookbehind the order stays be same, even though lookahead parentheses are before the main pattern.

Usually parentheses are numbered left-to-right, but lookbehind is an exception, it is always captured after the main pattern. So the match for \d+ goes in the result first, and then for (\\$/£).

Summary

Lookahead and lookbehind (commonly referred to as “lookaround”) are useful when we'd like to take something into the match depending on the context before/after it.

For simple regexps we can do the similar thing manually. That is: match everything, in any context, and then filter by context in the loop.

Remember, `str.matchAll` and `reg.exec` return matches with `.index` property, so we know where exactly in the text it is, and can check the context.

But generally regular expressions are more convenient.

Lookaround types:

Pattern	type	matches
<u>x(?=y)</u>	Positive lookahead	x if followed by y
<u>x(?!y)</u>	Negative lookahead	x if not followed by y
<u>(?<=y)x</u>	Positive lookbehind	x if after y
<u>(?<!y)x</u>	Negative lookbehind	x if not after y

Lookahead can also be used to disable backtracking. Why that may be needed and other details – see in the next chapter.

Infinite backtracking problem

Some regular expressions are looking simple, but can execute veeeeery long time, and even “hang” the JavaScript engine.

Sooner or later most developers occasionally face such behavior.

The typical situation – a regular expression works fine sometimes, but for certain strings it “hangs” consuming 100% of CPU.

In a web-browser it kills the page. Not a good thing for sure.

For server-side JavaScript it may become a vulnerability, and it uses regular expressions to process user data. Bad input will make the process hang, causing denial of service. The author personally saw and reported such vulnerabilities even for very well-known and widely used programs.

So the problem is definitely worth to deal with.

Introduction

The plan will be like this:

1. First we see the problem how it may occur.
2. Then we simplify the situation and see why it occurs.
3. Then we fix it.

For instance let’s consider searching tags in HTML.

We want to find all tags, with or without attributes – like ``. We need the regexp to work reliably, because HTML comes from the internet and can be messy.

In particular, we need it to match tags like `<a test="<>" href="#">` – with `<` and `>` in attributes. That’s allowed by [HTML standard](#).

A simple regexp like `<[>]+>` doesn’t work, because it stops at the first `>`, and we need to ignore `<>` if inside an attribute:

```
// the match doesn't reach the end of the tag - wrong!
alert( '<a test="<>" href="#">'.match(/<[>]+>/) ); // <a test="<>
```

To correctly handle such situations we need a more complex regular expression. It will have the form `<tag (key=value)*>`.

1. For the **tag name**: \w+,
2. For the **key name**: \w+,
3. And the **value**: a quoted string "[^"]*".

If we substitute these into the pattern above and throw in some optional spaces `\s`, the full regexp becomes: `<\w+(\s*\w+="[^"]*" \s*)*>`.

That regexp is not perfect! It doesn't support all the details of HTML syntax, such as unquoted values, and there are other ways to improve, but let's not add complexity. It will demonstrate the problem for us.

The regexp seems to work:

```
let reg = /<\w+(\\"s*\w+=\"[^\"]*"\"s*)*>/g;

let str='...<a test="<>" href="#">... <b>...';

alert( str.match(reg) ); // <a test="<>" href="#">, <b>
```

Great! It found both the long tag `<a test="<>" href="#">` and the short one ``.

Now, that we've got a seemingly working solution, let's get to the infinite backtracking itself.

Infinite backtracking

If you run our regexp on the input below, it may hang the browser (or another JavaScript host):

```
let reg = /<\w+(\s*\w+=("[^"]*"|'['']*'|[^'"]*)*)>/g;

let str = `<tag a="b"  a="b"  a="b"  a="b"  a="b"  a="b"  a="b"  a="b"
a="b"  a="b"  a="b"  a="b"  a="b"  a="b"  a="b"  a="b"  a="b"  a="b"  a="b"  a="b"

// The search will take a long, long time
alert( str.match(reg) );
```

Some regexp engines can handle that search, but most of them can't.

What's the matter? Why a simple regular expression “hangs” on such a small string?

Let's simplify the regexp by stripping the tag name and the quotes. So that we look only for `key=value` attributes: `<(\s*\w+=\w+\s*)*>`.

Unfortunately, the regexp still hangs:

```
// only search for space-delimited attributes
let reg = /<(\s*\w+=\w+\s*)*>/g;

let str = `<a=b a=b a=b a=b a=b a=b a=b a=b
a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b`;

// the search will take a long, long time
alert( str.match(reg) );
```

Here we end the demo of the problem and start looking into what's going on, why it hangs and how to fix it.

Detailed example

To make an example even simpler, let's consider `(\d+)*$`.

This regular expression also has the same problem. In most regexp engines that search takes a very long time (careful – can hang):

```
alert( '12345678901234567890123456789123456789z'.match(/(\d+)*$/) );
```

So what's wrong with the regexp?

First, one may notice that the regexp is a little bit strange. The quantifier `*` looks extraneous. If we want a number, we can use `\d+$`.

Indeed, the regexp is artificial. But the reason why it is slow is the same as those we saw above. So let's understand it, and then the previous example will become obvious.

What happens during the search of `(\d+)*$` in the line `123456789z`?

1. First, the regexp engine tries to find a number `\d+`. The plus `+` is greedy by default, so it consumes all digits:

```
\d+.....
(123456789)z
```

2. Then it tries to apply the star quantifier, but there are no more digits, so it the star doesn't give anything.
3. Then the pattern expects to see the string end `$`, and in the text we have `z`, so there's no match:

```
          X
\d+.....$
(123456789)z
```

4. As there's no match, the greedy quantifier + decreases the count of repetitions (backtracks).

Now \d+ doesn't take all digits, but all except the last one:

```
\d+.....
(12345678)9z
```

5. Now the engine tries to continue the search from the new position (9).

The star (\d+)* can be applied – it gives the number 9 :

```
\d+.....\d+
(12345678)(9)z
```

The engine tries to match \$ again, but fails, because meets z :

```
          X
\d+.....\d+
(12345678)(9)z
```

6. There's no match, so the engine will continue backtracking, decreasing the number of repetitions for \d+ down to 7 digits. So the rest of the string 89 becomes the second \d+ :

```
          X
\d+.....\d+
(1234567)(89)z
```

...Still no match for \$.

The search engine backtracks again. Backtracking generally works like this: the last greedy quantifier decreases the number of repetitions until it can. Then the previous greedy quantifier decreases, and so on. In our case the last greedy quantifier is the second \d+, from 89 to 8, and then the star takes 9 :

```

                X
\d+.....\d+\d+
(1234567)(8)(9)z
```

7. ...Fail again. The second and third `\d+` backtracked to the end, so the first quantifier shortens the match to 123456, and the star takes the rest:

```

                X
\d+.....\d+
(123456)(789)z
```

Again no match. The process repeats: the last greedy quantifier releases one character (9):

```

                X
\d+.....\d+ \d+
(123456)(78)(9)z
```

8. ...And so on.

The regular expression engine goes through all combinations of `123456789` and their subsequences. There are a lot of them, that's why it takes so long.

What to do?

Should we turn on the lazy mode?

Unfortunately, it doesn't: if we replace `\d+` with `\d+?`, that still hangs:

```
// slooooooooooooooooooooo
alert( '12345678901234567890123456789123456789z'.match(/(\d+?)*$/) );
```

Lazy quantifiers actually do the same, but in the reverse order.

Just think about how the search engine would work in this case.

Some regular expression engines have tricky built-in checks to detect infinite backtracking or other means to work around them, but there's no universal solution.

Back to tags

In the example above, when we search `<(\s*\w+=\w+\s*)*>` in the string `<a=b a=b a=b` – the similar thing happens.

The string has no `>` at the end, so the match is impossible, but the regexp engine doesn't know about it. The search backtracks trying different combinations of `(\s*\w+=\w+\s*)`:

```
(a=b a=b a=b) (a=b)
(a=b a=b) (a=b a=b)
(a=b) (a=b a=b a=b)
...
```

As there are many combinations, it takes a lot of time.

How to fix?

The backtracking checks many variants that are an obvious fail for a human.

For instance, in the pattern `(\d+)*$` a human can easily see that `(\d+)*` does not need to backtrack `+`. There's no difference between one or two `\d+`:

```
\d+.....
(123456789)z

\d+... \d+...
(1234)(56789)z
```

Let's get back to more real-life example: `<(\s*\w+=\w+\s*)*>`. We want it to find pairs `name=value` (as many as it can).

What we would like to do is to forbid backtracking.

There's totally no need to decrease the number of repetitions.

In other words, if it found three `name=value` pairs and then can't find `>` after them, then there's no need to decrease the count of repetitions. There are definitely no `>` after those two (we backtracked one `name=value` pair, it's there):

```
(name=value) name=value
```

Modern regexp engines support so-called “possessive” quantifiers for that. They are like greedy, but don't backtrack at all. Pretty simple, they capture whatever they can, and the search continues. There's also another tool called “atomic groups” that forbid backtracking inside parentheses.

Unfortunately, but both these features are not supported by JavaScript.

Lookahead to the rescue

We can forbid backtracking using lookahead.



The pattern to take as much repetitions as possible without backtracking is: `(?= (a+))\1`.

In other words:

- The lookahead `?=` looks for the maximal count `a+` from the current position.
- And then they are “consumed into the result” by the backreference `\1` (`\1` corresponds to the content of the second parentheses, that is `a+`).

There will be no backtracking, because lookahead does not backtrack. If, for example, it found 5 instances of `a+` and the further match failed, it won't go back to the 4th instance.

Please note:

There's more about the relation between possessive quantifiers and lookahead in articles [Regex: Emulate Atomic Grouping \(and Possessive Quantifiers\) with LookAhead](#)  and [Mimicking Atomic Groups](#) .

So this trick makes the problem disappear.

Let's fix the regexp for a tag with attributes from the beginning of the chapter `<\w+(\s*\w+(\w+|"[^"]*")\s*)*>`. We'll use lookahead to prevent backtracking of `name=value` pairs:

```
// regexp to search name=value
let attrReg = /(\s*\w+(\w+|"[^"]*" )\s*)*/

// use new RegExp to nicely insert its source into (?(a+))\1
let fixedReg = new RegExp(`<\w+(?=(${attrReg.source}*))\\1>`, 'g');

let goodInput = '...<a test="<>" href="#">... <b>...';

let badInput = `<tag a=b a=b a=b a=b a=b a=b a=b a=b
a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b a=b`;

alert( goodInput.match(fixedReg) ); // <a test="<>" href="#">, <b>
alert( badInput.match(fixedReg) ); // null (no results, fast!)
```

Great, it works! We found both a long tag `<a test="<>" href="#">` and a small one ``, and (!) didn't hang the engine on the bad input.

Please note the `attrReg.source` property. `RegExp` objects provide access to their source string in it. That's convenient when we want to insert one regexp into

another.

Unicode: flag "u"

The unicode flag `/.../u` enables the correct support of surrogate pairs.

Surrogate pairs are explained in the chapter [Strings](#).

Let's briefly review them here. In short, normally characters are encoded with 2 bytes. That gives us 65536 characters maximum. But there are more characters in the world.

So certain rare characters are encoded with 4 bytes, like `ℵ` (mathematical X) or `😊` (a smile).

Here are the unicode values to compare:

Character	Unicode	Bytes
a	0x0061	2
≈	0x2248	2
ℵ	0x1d4b3	4
ℶ	0x1d4b4	4
😊	0x1f604	4

So characters like `a` and `≈` occupy 2 bytes, and those rare ones take 4.

The unicode is made in such a way that the 4-byte characters only have a meaning as a whole.

In the past JavaScript did not know about that, and many string methods still have problems. For instance, `length` thinks that here are two characters:

```
alert('😊'.length); // 2
alert('ℵ'.length); // 2
```

...But we can see that there's only one, right? The point is that `length` treats 4 bytes as two 2-byte characters. That's incorrect, because they must be considered only together (so-called "surrogate pair").

Normally, regular expressions also treat "long characters" as two 2-byte ones.

That leads to odd results, for instance let's try to find `[ℵℶ]` in the string `ℵ`:

```
alert( 'ℵ'.match(/[\ℵℶ]/) ); // odd result (wrong match actually, "half-character"
```


The result is wrong, because by default the regexp engine does not understand surrogate pairs.

So, it thinks that `[XY]` are not two, but four characters:

1. the left half of `X` (1),
2. the right half of `X` (2),
3. the left half of `Y` (3),
4. the right half of `Y` (4).

We can list them like this:

```
for(let i=0; i<'XY'.length; i++) {  
  alert('XY'.charAt(i)); // 55349, 56499, 55349, 56500  
};
```

So it finds only the “left half” of `X`.

In other words, the search works like `'12'.match(/[1234]/)`: only `1` is returned.

The “u” flag

The `/.../u` flag fixes that.

It enables surrogate pairs in the regexp engine, so the result is correct:

```
alert( 'X'.match(/[XY]/u) ); // X
```

Let's see one more example.

If we forget the `u` flag and occasionally use surrogate pairs, then we can get an error:

```
'X'.match(/[X-Y]/); // SyntaxError: invalid range in character class
```

Normally, regexps understand `[a-z]` as a “range of characters with codes between codes of `a` and `z`”.


But without `u` flag, surrogate pairs are assumed to be a “pair of independent characters”, so `[X-Y]` is like `[<55349><56499>-<55349><56500>]` (replaced each surrogate pair with code points). Now we can clearly see that the

range 56499-55349 is unacceptable, as the left range border must be less than the right one.

Using the `u` flag makes it work right:

```
alert( 'y'.match(/[x-z]/u) ); // y
```

Unicode character properties \p

[Unicode](#) , the encoding format used by JavaScript strings, has a lot of properties for different characters (or, technically, code points). They describe which “categories” character belongs to, and a variety of technical details.


In regular expressions these can be set by `\p{...}`. And there must be flag `'u'`.

For instance, `\p{Letter}` denotes a letter in any of language. We can also use `\p{L}`, as `L` is an alias of `Letter`, there are shorter aliases for almost every property.


Here's the main tree of properties:

- Letter `L`:
 - lowercase `Ll`, modifier `Lm`, titlecase `Lt`, uppercase `Lu`, other `Lo`
- Number `N`:
 - decimal digit `Nd`, letter number `Nl`, other `No`
- Punctuation `P`:
 - connector `Pc`, dash `Pd`, initial quote `Pi`, final quote `Pf`, open `Ps`, close `Pe`, other `Po`
- Mark `M` (accents etc):
 - spacing combining `Mc`, enclosing `Me`, non-spacing `Mn`
- Symbol `S`:
 - currency `Sc`, modifier `Sk`, math `Sm`, other `So`
- Separator `Z`:
 - line `Zl`, paragraph `Zp`, space `Zs`
- Other `C`:
 - control `Cc`, format `Cf`, not assigned `Cn`, private use `Co`, surrogate `Cs`

More information

Interested to see which characters belong to a property? There's a tool at <http://cldr.unicode.org/unicode-utilities/list-unicode>  for that.

You could also explore properties at [Character Property Index](#) .

For the full Unicode Character Database in text format (along with all properties), see <https://www.unicode.org/Public/UCD/latest/ucd/> .

There are also other derived categories, like:

- `Alphabetic` (`Alpha`), includes Letters `L`, plus letter numbers `NI` (e.g. roman numbers `XII`), plus some other symbols `Other_Alphabetic` (`OAItpa`).
- `Hex_Digit` includes hexadimal digits: `0-9`, `a-f`.
- ...Unicode is a big beast, it includes a lot of properties.

For instance, let's look for a 6-digit hex number:

```
let reg = /\p{Hex_Digit}{6}/u; // flag 'u' is required
alert("color: #123ABC".match(reg)); // 123ABC
```

There are also properties with a value. For instance, Unicode “Script” (a writing system) can be Cyrillic, Greek, Arabic, Han (Chinese) etc, the [list is long](#).

To search for characters in certain scripts (“alphabets”), we should supply `Script=<value>`, e.g. to search for cyrillic letters: `\p{sc=Cyrillic}`, for Chinese glyphs: `\p{sc=Han}`, etc:

```
let regexp = /\p{sc=Han}+/gu; // get chinese words
let str = `Hello Привет 你好 123_456`;
alert( str.match(regexp) ); // 你好
```

Building multi-language `\w`

The pattern `\w` means “wordly characters”, but doesn't work for languages that use non-Latin alphabets, such as Cyrillic and others. It's just a shorthand for `[a-zA-Z0-9_]`, so `\w+` won't find any Chinese words etc.

Let's make a “universal” regexp, that looks for wordly characters in any language. That's easy to do using Unicode properties:

```
/[\p{Alphabetic}\p{Mark}\p{Decimal_Number}\p{Connector_Punctuation}\p{Join_Control}]
```

Let's decipher. Just as `\w` is the same as `[a-zA-Z0-9_]`, we're making a set of our own, that includes:

- `Alphabetic` for letters,
- `Mark` for accents, as in Unicode accents may be represented by separate code points,
- `Decimal_Number` for numbers,
- `Connector_Punctuation` for the `'_'` character and alike,
- `Join_Control` — two special code points with hex codes `200c` and `200d`, used in ligatures e.g. in arabic.

Or, if we replace long names with aliases (a list of aliases [here](#)):

```
let regexp = /([\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]+)/gu;

let str = `Hello Привет 你好 123_456`;

alert( str.match(regexp) ); // Hello,Привет,你好,123_456
```

Sticky flag "y", searching at position

To grasp the use case of `y` flag, and see how great it is, let's explore a practical use case.

One of common tasks for regexps is “parsing”: when we get a text and analyze it for logical components, build a structure.

For instance, there are HTML parsers for browser pages, that turn text into a structured document. There are parsers for programming languages, like JavaScript, etc.

Writing parsers is a special area, with its own tools and algorithms, so we don't go deep in there, but there's a very common question in them, and, generally, for text analysis: “What kind of entity is at the given position?”.

For instance, for a programming language variants can be like:

- Is it a “name” `\w+`?
- Or is it a number `\d+`?
- Or an operator `[+ - / *]`?
- (a syntax error if it's not anything in the expected list)

So, we should try to match a couple of regular expressions, and make a decision what's at the given position.

In JavaScript, how can we perform a search starting from a given position? Regular calls start searching from the text start.

We'd like to avoid creating substrings, as this slows down the execution considerably.

One option is to use `regexp.exec` with `regexp.lastIndex` property, but that's not what we need, as this would search the text starting from `lastIndex`, while we only need to test the match *exactly* at the given position.

Here's a (failing) attempt to use `lastIndex`:

```
let str = "(text before) function ...";

// attempting to find function at position 5:
let regexp = /function/g; // must use "g" flag, otherwise lastIndex is ignored
regexp.lastIndex = 5

alert (regexp.exec(str)); // function
```

The match is found, because `regexp.exec` starts to search from the given position and goes on by the text, successfully matching "function" later.

We could work around that by checking if `regexp.exec(str).index` property is `5`, and if not, ignore the match. But the main problem here is performance. The regexp engine does a lot of unnecessary work by scanning at further positions. The delays are clearly noticeable if the text is long, because there are many such searches in a parser.

The "y" flag

So we've come to the problem: how to search for a match exactly at the given position.

That's what `y` flag does. It makes the regexp search only at the `lastIndex` position.

Here's an example

```
let str = "(text before) function ...";

let regexp = /function/y;
regexp.lastIndex = 5;

alert (regexp.exec(str)); // null (no match, unlike "g" flag!)
```

```
regex.lastIndex = 14;  
  
alert (regex.exec(str)); // function (match!)
```

As we can see, now the regexp is only matched at the given position.

So what `y` does is truly unique, and very important for writing parsers.

The `y` flag allows to test a regular expression exactly at the given position and when we understand what's there, we can move on – step by step examining the text.

Without the flag the regexp engine always searches till the end of the text, that takes time, especially if the text is large. So our parser would be very slow. The `y` flag is exactly the right thing here.

Solutions

ArrayBuffer, binary arrays

Concatenate typed arrays

```
function concat(arrays) {  
  // sum of individual array lengths  
  let totalLength = arrays.reduce((acc, value) => acc + value.length, 0);  
  
  if (!arrays.length) return null;  
  
  let result = new Uint8Array(totalLength);  
  
  // for each array - copy it over result  
  // next array is copied right after the previous one  
  let length = 0;  
  for(let array of arrays) {  
    result.set(array, length);  
    length += array.length;  
  }  
  
  return result;  
}
```

[Open the solution with tests in a sandbox.](#) ↗

To formulation

Fetch

Fetch users from GitHub

To fetch a user we need:

1. `fetch('https://api.github.com/users/USERNAME')`.
2. If the response has status `200`, call `.json()` to read the JS object.

If a `fetch` fails, or the response has non-200 status, we just return `null` in the resulting array.

So here's the code:

```
async function getUsers(names) {
  let jobs = [];

  for(let name of names) {
    let job = fetch(`https://api.github.com/users/${name}`).then(
      successResponse => {
        if (successResponse.status !== 200) {
          return null;
        } else {
          return successResponse.json();
        }
      },
      failResponse => {
        return null;
      }
    );
    jobs.push(job);
  }

  let results = await Promise.all(jobs);

  return results;
}
```

Please note: `.then` call is attached directly to `fetch`, so that when we have the response, it doesn't wait for other fetches, but starts to read `.json()` immediately.

If we used `await Promise.all(names.map(name => fetch(...)))`, and call `.json()` on the results, then it would wait for all fetches to respond. By adding `.json()` directly to each `fetch`, we ensure that individual fetches start reading data as JSON without waiting for each other.

That's an example of how low-level `Promise` API can still be useful even if we mainly use `async/await`.

[Open the solution with tests in a sandbox.](#) ↗

[To formulation](#)

Fetch: Cross-Origin Requests

Why do we need Origin?

We need `Origin`, because sometimes `Referer` is absent. For instance, when we `fetch` HTTP-page from HTTPS (access less secure from more secure), then there's no `Referer`.

The [Content Security Policy](#) ↗ may forbid sending a `Referer`.

As we'll see, `fetch` also has options that prevent sending the `Referer` and even allow to change it (within the same site).

By specification, `Referer` is an optional HTTP-header.

Exactly because `Referer` is unreliable, `Origin` was invented. The browser guarantees correct `Origin` for cross-origin requests.

[To formulation](#)

LocalStorage, sessionStorage

Autosave a form field

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

CSS-animations

Animate a plane (CSS)

CSS to animate both `width` and `height`:

```
/* original class */

#flyjet {
  transition: all 3s;
}

/* JS adds .growing */
#flyjet.growing {
  width: 400px;
  height: 240px;
}
```

Please note that `transitionend` triggers two times – once for every property. So if we don't perform an additional check then the message would show up 2 times.

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

Animate the flying plane (CSS)

We need to choose the right Bezier curve for that animation. It should have `y>1` somewhere for the plane to “jump out”.

For instance, we can take both control points with `y>1`, like: `cubic-bezier(0.25, 1.5, 0.75, 1.5)`.

The graph:



[Open the solution in a sandbox.](#) ↗

[To formulation](#)

Animated circle

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

JavaScript animations

Animate the bouncing ball

To bounce we can use CSS property `top` and `position:absolute` for the ball inside the field with `position:relative`.

The bottom coordinate of the field is `field.clientHeight`. The CSS `top` property refers to the upper edge of the ball. So it should go from `0` till `field.clientHeight - ball.clientHeight`, that's the final lowest position of the upper edge of the ball.

To to get the “bouncing” effect we can use the timing function `bounce` in `easeOut` mode.

Here's the final code for the animation:

```
let to = field.clientHeight - ball.clientHeight;

animate({
  duration: 2000,
  timing: makeEaseOut(bounce),
  draw(progress) {
    ball.style.top = to * progress + 'px'
  }
});
```

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

Animate the ball bouncing to the right

In the task [Animate the bouncing ball](#) we had only one property to animate. Now we need one more: `elem.style.left`.

The horizontal coordinate changes by another law: it does not “bounce”, but gradually increases shifting the ball to the right.

We can write one more `animate` for it.

As the time function we could use `linear`, but something like `makeEaseOut(quad)` looks much better.

The code:

```
let height = field.clientHeight - ball.clientHeight;
let width = 100;

// animate top (bouncing)
animate({
  duration: 2000,
  timing: makeEaseOut(bounce),
  draw: function(progress) {
    ball.style.top = height * progress + 'px'
  }
});

// animate left (moving to the right)
animate({
  duration: 2000,
  timing: makeEaseOut(quad),
  draw: function(progress) {
    ball.style.left = width * progress + "px"
  }
});
```

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

Custom elements

Live timer element

Please note:

1. We clear `setInterval` timer when the element is removed from the document. That's important, otherwise it continues ticking even if not needed any more. And the browser can't clear the memory from this element and referenced by it.
2. We can access current date as `elem.date` property. All class methods and properties are naturally element methods and properties.

[Open the solution in a sandbox.](#) ↗

[To formulation](#)

Character classes

Find the time

The answer: `\b\d\d:\d\d\b.`

```
alert( "Breakfast at 09:00 in the room 123:456.".match( /\b\d\d:\d\d\b/ ) );
```

[To formulation](#)

Sets and ranges [...]

Java[^script]

Answers: **no**, **yes**.

- In the script `Java` it doesn't match anything, because `[^script]` means "any character except given ones". So the regexp looks for `"Java"` followed by one such symbol, but there's a string end, no symbols after it.

```
alert( "Java".match(/Java[^script]/) ); // null
```

- Yes, because the regexp is case-insensitive, the `[^script]` part matches the character `"S"`.

```
alert( "JavaScript".match(/Java[^script]/) ); // "JavaS"
```

To formulation

Find the time as hh:mm or hh-mm

Answer: \d\d[-:]\d\d.

```
let reg = /\d\d[-:]\d\d/g;
alert( "Breakfast at 09:00. Dinner at 21-30".match(reg) ); // 09:00, 21-30
```

Please note that the dash ' - ' has a special meaning in square brackets, but only between other characters, not when it's in the beginning or at the end, so we don't need to escape it.

To formulation

Quantifiers +, *, ? and {n}

How to find an ellipsis "..." ?

Solution:

```
let reg = /\.{3,}/g;
alert( "Hello!... How goes?.....".match(reg) ); // ..., .....
```

Please note that the dot is a special character, so we have to escape it and insert as \..

To formulation

Regexp for HTML colors

We need to look for # followed by 6 hexadecimal characters.

A hexadecimal character can be described as [0-9a-fA-F]. Or if we use the i flag, then just [0-9a-f].

Then we can look for 6 of them using the quantifier {6}.

As a result, we have the regexp: /#[a-f0-9]{6}/gi.

```
let reg = /#[a-f0-9]{6}/gi;

let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2"

alert( str.match(reg) ); // #121212,#AA00ef
```

The problem is that it finds the color in longer sequences:

```
alert( "#12345678".match( /#[a-f0-9]{6}/gi ) ) // #12345678
```

To fix that, we can add \b to the end:

```
// color
alert( "#123456".match( /#[a-f0-9]{6}\b/gi ) ); // #123456

// not a color
alert( "#12345678".match( /#[a-f0-9]{6}\b/gi ) ); // null
```

[To formulation](#)

Greedy and lazy quantifiers

A match for /d+? d+?/

The result is: 123 4.

First the lazy \d+? tries to take as little digits as it can, but it has to reach the space, so it takes 123.

Then the second \d+? takes only one digit, because that's enough.

[To formulation](#)

Find HTML comments

We need to find the beginning of the comment <!--, then everything till the end of -->.

The first idea could be `<!--.*?-->` – the lazy quantifier makes the dot stop right before `-->`.

But a dot in JavaScript means “any symbol except the newline”. So multiline comments won’t be found.

We can use `[\s\S]` instead of the dot to match “anything”:

```
let reg = /<!--[\s\S]*?-->/g;

let str = `... <!-- My -- comment
test --> .. <!------> ..
`;

alert( str.match(reg) ); // '<!-- My -- comment \n test -->', '<!------>'
```

To formulation

Find HTML tags

The solution is `<[<>]+>`.

```
let reg = /<[<>]+>/g;

let str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(reg) ); // '<a href="/">', '<input type="radio" checked>'
```

To formulation

Capturing groups

Find color in the format #abc or #abcdef

A regexp to search 3-digit color `#abc`: `/#[a-f0-9]{3}/i`.

We can add exactly 3 more optional hex digits. We don’t need more or less. Either we have them or we don’t.

The simplest way to add them – is to append to the regexp: `/#[a-f0-9]{3}([a-f0-9]{3})?/i`

We can do it in a smarter way though: /#([a-f0-9]{3}){1,2}/i.

Here the regexp [a-f0-9]{3} is in parentheses to apply the quantifier {1,2} to it as a whole.

In action:

```
let reg = /#([a-f0-9]{3}){1,2}/gi;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(reg) ); // #3f3 #AA00ef #abc
```

There's a minor problem here: the pattern found #abc in #abcd. To prevent that we can add \b to the end:

```
let reg = /#([a-f0-9]{3}){1,2}\b/gi;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(reg) ); // #3f3 #AA00ef
```

To formulation

Find positive numbers

An non-negative integer number is \d+. A zero 0 can't be the first digit, but we should allow it in further digits.

So that gives us [1-9]\d*.

A decimal part is: \.\d+.

Because the decimal part is optional, let's put it in parentheses with the quantifier ?.

Finally we have the regexp: [1-9]\d*(\.\d+)?:

```
let reg = /[1-9]\d*(\.\d+)?/g;

let str = "1.5 0 -5 12. 123.4.";

alert( str.match(reg) ); // 1.5, 0, 12, 123.4
```


Find all numbers

A positive number with an optional decimal part is (per previous task): `\d+(\.\d+)?`.

Let's add an optional `-` in the beginning:

```
let reg = /-?\d+(\.\d+)?/g;

let str = "-1.5 0 2 -123.4.";

alert( str.match(reg) );    // -1.5, 0, 2, -123.4
```

Parse an expression

A regexp for a number is: `-?\d+(\.\d+)?`. We created it in previous tasks.

An operator is `[-+*/]`.

Please note:

- Here the dash `-` goes first in the brackets, because in the middle it would mean a character range, while we just want a character `-`.
- A slash `/` should be escaped inside a JavaScript regexp `/.../`, we'll do that later.

We need a number, an operator, and then another number. And optional spaces between them.

The full regular expression: `-?\d+(\.\d+)?\s*[-+*/]\s*-?\d+(\.\d+)?`.

To get a result as an array let's put parentheses around the data that we need: numbers and the operator: `(-?\d+(\.\d+)?)\s*([-+*/])\s*(-?\d+(\.\d+)?)`.

In action:

```
let reg = /(-?\d+(\.\d+)?)\s*([-\+*\\/])\s*(-?\d+(\.\d+)?)/;

alert( "1.2 + 12".match(reg) );
```

The result includes:

- `result[0] == "1.2 + 12"` (full match)
- `result[1] == "1.2"` (first group `(-?\d+(\.\d+)?)` – the first number, including the decimal part)
- `result[2] == ".2"` (second group `(\.\d+)?` – the first decimal part)
- `result[3] == "+"` (third group `([-\+*\\/])` – the operator)
- `result[4] == "12"` (forth group `(-?\d+(\.\d+)?)` – the second number)
- `result[5] == undefined` (fifth group `(\.\d+)?` – the last decimal part is absent, so it's undefined)

We only want the numbers and the operator, without the full match or the decimal parts.

The full match (the arrays first item) can be removed by shifting the array `result.shift()`.

The decimal groups can be removed by making them into non-capturing groups, by adding `?:` to the beginning: `(?:\.\d+)?`.

The final solution:

```
function parse(expr) {
  let reg = /(-?\d+(?:\.\d+)?)\s*([-\+*\\/])\s*(-?\d+(?:\.\d+)?)/;

  let result = expr.match(reg);

  if (!result) return [];
  result.shift();

  return result;
}

alert( parse("-1.23 * 3.45") ); // -1.23, *, 3.45
```

To formulation

Alternation (OR) |

Find programming languages

The first idea can be to list the languages with `|` in-between.

But that doesn't work right:

```
let reg = /Java|JavaScript|PHP|C|C\+\+/g;

let str = "Java, JavaScript, PHP, C, C++";

alert( str.match(reg) ); // Java,Java,PHP,C,C
```

The regular expression engine looks for alternations one-by-one. That is: first it checks if we have Java, otherwise – looks for JavaScript and so on.

As a result, JavaScript can never be found, just because Java is checked first.

The same with C and C++.

There are two solutions for that problem:

1. Change the order to check the longer match first:

JavaScript | Java | C\+\+ | C | PHP.

2. Merge variants with the same start: Java(Script)? | C(\+\+)?
| PHP.

In action:

```
let reg = /Java(Script)?|C(\+\+)?|PHP/g;

let str = "Java, JavaScript, PHP, C, C++";

alert( str.match(reg) ); // Java,JavaScript,PHP,C,C++
```

To formulation

Find bbtag pairs

Opening tag is \[(b|url|quote)\].

Then to find everything till the closing tag – let's use the pattern `. *?` with flag `s` to match any character including the newline and then add a backreference to the closing tag.

The full pattern: `\[(b|url|quote)\]. *?\[/\1\]`.

In action:

```
let reg = /\[(b|url|quote)\]. *?\[/\1\]/gs;

let str = `
  [b]hello![/b]
  [quote]
    [url]http://google.com[/url]
  [/quote]
`;

alert( str.match(reg) ); // [b]hello![/b],[quote][url]http://google.com[/url]
```

Please note that we had to escape a slash for the closing tag `\[/\1\]`, because normally the slash closes the pattern.

To formulation

Find quoted strings

The solution: `/"(\\.|[^\\""])*"/g`.

Step by step:

- First we look for an opening quote `"`
- Then if we have a backslash `\\` (we technically have to double it in the pattern, because it is a special character, so that's a single backslash in fact), then any character is fine after it (a dot).
- Otherwise we take any character except a quote (that would mean the end of the string) and a backslash (to prevent lonely backslashes, the backslash is only used with some other symbol after it): `[^\\""]`
- ...And so on till the closing quote.

In action:

```
let reg = /"(\\.|[^\\""])*"/g;
let str = ' .. "test me" .. "Say \\\"Hello\\\"!" .. "\\\\" .. ' ';

alert( str.match(reg) ); // "test me","Say \\\"Hello\\\"!", "\\\""
```

To formulation

Find the full tag

The pattern start is obvious: <style.

...But then we can't simply write <style.*?>, because <styler> would match it.

We need either a space after <style and then optionally something else or the ending >.

In the regexp language: <style(>|\s.*?>).

In action:

```
let reg = /<style(>|\s.*?>)/g;

alert( '<style> <styler> <style test="...">'.match(reg) ); // <style>, <styl
```

To formulation

String start ^ and finish \$

Regexp ^\$

The empty string is the only match: it starts and immediately finishes.

The task once again demonstrates that anchors are not characters, but tests.

The string is empty `""`. The engine first matches the ^ (input start), yes it's there, and then immediately the end \$, it's here too. So there's a match.

To formulation

Check MAC-address

A two-digit hex number is [0-9a-f]{2} (assuming the i flag is enabled).

We need that number `NN`, and then `:NN` repeated 5 times (more numbers);

The regexp is: `[0-9a-f]{2}(:[0-9a-f]{2}){5}`

Now let's show that the match should capture all the text: start at the beginning and end at the end. That's done by wrapping the pattern in `^...$`.

Finally:

```
let reg = /^[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}$/i;

alert( reg.test('01:32:54:67:89:AB') ); // true

alert( reg.test('0132546789AB') ); // false (no colons)

alert( reg.test('01:32:54:67:89') ); // false (5 numbers, need 6)

alert( reg.test('01:32:54:67:89:ZZ') ); // false (ZZ in the end)
```

To formulation