TutorialFrames and windows{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},{"@type":"ListItem","position":2,"name":"Frames and windows","item":"https://javascript.info/frames-and-windows"}]}November 4, 2022Popups and window methodsA 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 url in 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:

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

Popup blockingIn 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.

window.openThe 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.
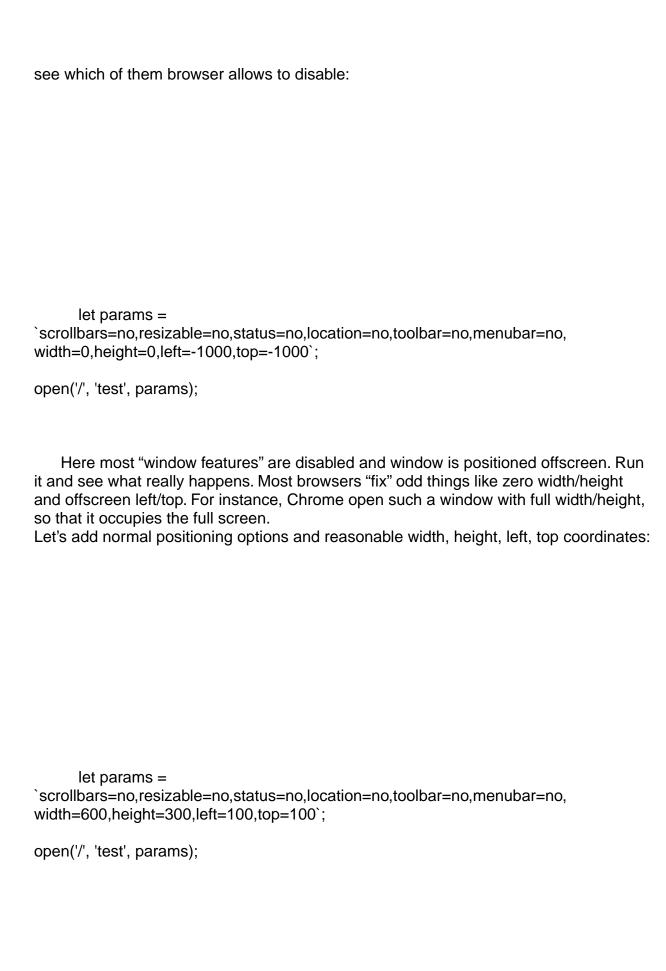
Window features:

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 windowLet'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 windowThe open call returns a reference to the new window. It can be used to manipulate its 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(newWindow.location.href); // (*) about:blank, loading hasn't started yet

newWindow.onload = function() {
  let html = `<div style="font-size:30px">Welcome!</div>`;
```

```
  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 popupA 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 popupTo 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
};
```

Moving and resizingThere 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 windowWe 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.scrollIntoView(top = true)
Scroll the window to make elem show up at the top (the default) or at the bottom for elem.scrollIntoView(false).

There's also window.onscroll event.
Focus/blur on a windowTheoretically, there are window.focus() and window.blur() methods to focus/unfocus on a window. And there are also focus/blur events that allow to catch the moment when the visitor focuses on a window and switches elsewhere. Although, in practice they are severely limited, because in the past evil pages abused them.
For instance, look at this code:

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

When a user attempts to switch out of the window (window.onblur), it brings the window back into focus. The intention is to "lock" the user within the window.
So browsers had to introduce many limitations to forbid the code like that and protect the user from ads and evils pages. They depend on the browser.
For instance, a mobile browser usually ignores window.focus() completely. Also focusing doesn't work when a popup opens in a separate tab rather than a new window.
Still, there are some use cases when such calls do work and can be useful.

For instance:

When we open a popup, it might be a good idea to run 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.

SummaryPopup 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.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/popup-windows","identifier":"\/ popup-windows"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialFrames and windows{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement":

[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Frames and windows","item":"https:// javascript.info/frames-and-windows"}]}April 13, 2022Cross-window communicationThe "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 OriginTwo 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: iframeAn <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>, shorthand for 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:

```html
<!-- 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.domainBy 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:

```js
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.

Deprecated, but still working
The document.domain property is in the process of being removed from the specification. The cross-window messaging (explained soon below) is the suggested replacement.
That said, as of now all browsers support it. And the support will be kept for the future, not to break old code that relies on document.domain.

Iframe: wrong document pitfallWhen an iframe comes from the same origin, and we may access its document, there's a pitfall. It's not related to cross-origin 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.framesAn 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 attributeThe 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:
Resultindex.htmlsandboxed.html<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
</head>

<body>

  <div>The iframe below has the <code>sandbox</code> attribute.</div>

  <iframe sandbox src="sandboxed.html" style="height:60px;width:90%"></iframe>

</body>
</html><!doctype html>
<html>

<head>
  <meta charset="UTF-8">
</head>

<body>

  <button onclick="alert(123)">Click to run a script (doesn't work)</button>

  <form action="http://google.com">
    <input type="text">
    <input type="submit" value="Submit (doesn't work)">
  </form>

</body>
</html>
        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 messagingThe 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.
postMessageThe 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 serialization 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>
```

onmessageTo 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:
Resultiframe.htmlindex.html<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
</head>

<body>

  Receiving iframe.

```
  <script>
    window.addEventListener('message', function(event) {
      alert(`Received ${event.data} from ${event.origin}`);
    });
  </script>

</body>
</html>
```

```
<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
</head>

<body>

  <form id="form">
    <input type="text" placeholder="Enter message" name="message">
    <input type="submit" value="Click to send">
  </form>

  <iframe src="iframe.html" id="iframe" style="display:block;height:60px"></iframe>

  <script>
    form.onsubmit = function() {
      iframe.contentWindow.postMessage(this.message.value, '*');
      return false;
    };
  </script>

</body>
</html>
```

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:

The sender calls targetWin.postMessage(data, targetOrigin).

If targetOrigin is not '*', then the browser checks if window targetWin has the origin targetOrigin.

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.

[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Frames and windows","item":"https:// javascript.info/frames-and-windows"}]}April 25, 2022The clickjacking attackThe "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 ideaThe idea is very simple.

Here's how clickjacking was done with Facebook:

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

The page has a harmless-looking link on it (like "get rich now" or "click here, very funny").

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.

In attempting to click the link, the visitor in fact clicks the button.

The demoHere'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:
Resultfacebook.htmlindex.html<!DOCTYPE HTML>
<html>

```
<body style="margin:10px;padding:10px">

  <input type="button" onclick="alert('Like pressed on facebook.html!')" value="I LIKE
IT !">

</body>

</html><!doctype html>
<html>

<head>
  <meta charset="UTF-8">
</head>

<body>

  <style>
    iframe {
      width: 400px;
      height: 100px;
      position: absolute;
      top: 5px;
      left: -14px;
      opacity: 0.5;
      z-index: 1;
    }
  </style>

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

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

  <button>Click here!</button>
```

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

</body>
</html>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>:
Resultfacebook.htmlindex.html<!DOCTYPE HTML>
<html>

<body style="margin:10px;padding:10px">

  <input type="button" onclick="alert('Like pressed on facebook.html!')" value="I LIKE IT !">

</body>

</html><!doctype html>
<html>

<head>
  <meta charset="UTF-8">
</head>

<body>

  <style>
    iframe {
      width: 400px;
      height: 100px;
      position: absolute;
      top: 5px;
      left: -14px;
      opacity: 0;
      z-index: 1;
    }
  </style>

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

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

  <button>Click here!</button>

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

</body>
</html>
```
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.

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:

```javascript
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-navigationWe 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:

```javascript
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:

Resultiframe.htmlindex.html

```html
<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
</head>

<body>

  <div>Changes top.location to javascript.info</div>

  <script>
    top.location = 'https://javascript.info';
  </script>

</body>

</html>
```

```html
<!doctype html>
<html>

<head>
  <meta charset="UTF-8">

  <style>
    iframe {
      width: 400px;
      height: 100px;
      position: absolute;
      top: 0;
      left: -20px;
      opacity: 0;
      z-index: 1;
    }
  </style>

  <script>
```

```
  function attack() {

    window.onbeforeunload = function() {
      window.onbeforeunload = null;
      return "Want to leave without learning all the secrets (he-he)?";
    };

    document.body.insertAdjacentHTML('beforeend', '<iframe src="iframe.html">');
  }
 </script>
</head>

<body>

 <p>After a click on the button the visitor gets a "strange" question about whether they
want to leave.</p>

 <p>Probably they would respond "No", and the iframe protection is hacked.</p>

 <button onclick="attack()">Add a "protected" iframe</button>

</body>
</html>
```

Sandbox attributeOne 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-OptionsThe 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.
Here's the result:

```html
<iframe src="https://twitter.com"></iframe>
```

Depending on your browser, the iframe above is either empty or alerting you that the browser won't permit that page to be navigating in this way.
Showing with disabled functionalityThe 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:

```html
    <style>
  #protector {
    height: 100%;
    width: 100%;
    position: absolute;
    left: 0;
    top: 0;
    z-index: 99999999;
  }
</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:

Resultiframe.htmlindex.html<!doctype html>
<html>

```html
<head>
  <meta charset="UTF-8">

  <style>
    #protector {
      height: 100%;
      width: 100%;
      position: absolute;
      left: 0;
      top: 0;
      z-index: 99999999;
    }
  </style>

</head>

<body>

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

<script>

  if (top.document.domain == document.domain) {
    protector.remove();
  }

</script>
```

This text is always visible.

But if the page was open inside a document from another domain, the div over it would prevent any actions.

```
  <button onclick="alert(1)">Click wouldn't work in that case</button>

</body>
</html><!doctype html>
<html>

<head>
  <meta charset="UTF-8">
</head>
<body>

  <iframe src="iframe.html"></iframe>

</body>
</html>
```
Samesite cookie attributeThe 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.

SummaryClickjacking 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.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/clickjacking","identifier":"\/clickjacking"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; Tutorial{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}]}Binary data, filesWorking with binary data and files in JavaScript.
ArrayBuffer, binary arraysTextDecoder and TextEncoderBlobFile and FileReaderCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial map
TutorialBinary data, files{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Binary data, files","item":"https://javascript.info/binary"}]}July 11, 2022ArrayBuffer, binary arraysIn 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 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 5.0x10-324 to 1.8x10308.

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).

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)
}
```

TypedArrayThe common term for all these views (Uint8Array, Uint32Array, etc) is
TypedArray. They share the same set of methods and properties.
Please note, there's no constructor called TypedArray, it's just a common "umbrella"
term to represent one of views over ArrayBuffer: Int8Array, Uint8Array and so on, the
full list will soon follow.
When you see something like new TypedArray, it means any of new Int8Array, new
Uint8Array, etc.
Typed arrays behave like regular arrays: have indexes and are 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();
```

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.

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 values
```

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 (explanations below)
```

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)
```

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 underlying ArrayBuffer, there are following properties in TypedArray:

buffer – references the ArrayBuffer.
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 behaviorWhat 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:

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:

In other words, the number modulo 2⁸ 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 methodsTypedArray 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.
DataViewDataView 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 65535
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. For example, when we store a sequence of pairs (16-bit integer, 32-bit float), DataView allows to access them easily.
SummaryArrayBuffer 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 denominator". 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:

TasksConcatenate typed arraysGiven an array of Uint8Array, write a function concat(arrays) that returns a concatenation of them into a single array.
Open a sandbox with tests.solution

```
function concat(arrays) {
// sum of individual array lengths
let totalLength = arrays.reduce((acc, value) => acc + value.length, 0);

let result = new Uint8Array(totalLength);

if (!arrays.length) return result;

// 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.Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:VV javascript.infoVarraybuffer-binary-arrays","identifier":"Varraybuffer-binary-arrays"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialBinary data, files{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Binary data, files","item":"https://javascript.info/
binary"}]}August 21, 2021TextDecoder and TextEncoderWhat if the binary data is actually a string? For instance, we received a file with textual data.

The built-in TextDecoder object allows one 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) ); // O`Y}
```

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

TextEncoderTextEncoder 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
```

BlobArrayBuffer 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 BufferSource.

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.

    Blob objects are immutable
    We can't change data directly in a Blob, but we can slice parts of a Blob, create
new Blob objects 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 URLA Blob can be easily used as a URL for <a>, <img> or other tags, to show its contents.
Thanks to type, we can also download/upload Blob objects, and the type 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

For each URL generated by URL.createObjectURL the browser stores a 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 <img>, <a>, basically any other object that expects a URL.
There's a side effect though. While there's a 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 Blob objects are freed then. But if an app is long-living, then that doesn't happen soon.
So if we create a 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 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 base64An 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 par with "regular" urls.

For instance, here's a smiley:

```
<img src="data:image/png;base64,R0lGODlhDAAMAKIFAF5LAP/zxAAAANyuAP/
gaP///wAAAAAACH5BAEAAAUALAAAAAAMAAwAAAMlWLPcGjDKFYi9lxKBOaGcF3
5DhWHamZUW0K4mAbiwWtuf0uxFAgA7">
```

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 a 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 urlNo need to revoke anything.
Performance and memory losses on big Blob objects for encoding.
Image to blobWe 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:

Draw an image (or its part) on canvas using canvas.drawImage.
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 ArrayBufferThe 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 blob.arrayBuffer():

```
    // get arrayBuffer from blob
const bufferPromise = await blob.arrayBuffer();

// or
blob.arrayBuffer().then(buffer => /* process the ArrayBuffer */);
```

From Blob to streamWhen we read and write to a blob of more than 2 GB, the use of arrayBuffer becomes more memory intensive for us. At this point, we can directly convert the blob to a stream.

A stream is a special object that allows to read from it (or write into it) portion by portion. It's outside of our scope here, but here's an example, and you can read more at https://developer.mozilla.org/en-US/docs/Web/API/Streams_API. Streams are convenient for data that is suitable for processing piece-by-piece.

The Blob interface's stream() method returns a ReadableStream which upon reading returns the data contained within the Blob.

Then we can read from it, like this:

```
        // get readableStream from blob
const readableStream = blob.stream();
const stream = readableStream.getReader();

while (true) {
  // for each iteration: value is the next blob fragment
  let { done, value } = await stream.read();
  if (done) {
    // no more data in the stream
    console.log('all blob processed.');
    break;
  }

  // do something with the data portion we've just read from the blob
  console.log(value);
}
```

SummaryWhile 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 blob.arrayBuffer(), and then create a view over it for low-level binary processing.

Conversion streams are very useful when we need to handle large blob. You can easily create a ReadableStream from a blob. The Blob interface's stream() method returns a ReadableStream which upon reading returns the data contained within the blob.

commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:VVjavascript.infoVblob","identifier":"Vblob"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialBinary data, files{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Binary data, files","item":"https://javascript.info/ binary"}]}April 9, 2020File and FileReaderA 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>
```

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].

FileReaderFileReader 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>
```

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.

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.

SummaryFile 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 <img>. 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.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/file","identifier":"\/file"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; Tutorial{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}]} Network requestsFetchFormDataFetch: Download progressFetch: AbortFetch: Cross-Origin RequestsFetch APIURL objectsXMLHttpRequestResumable file uploadLong pollingWebSocketServer Sent EventsCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial map

TutorialNetwork requests{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/ network"}]}April 14, 2022FetchJavaScript can send network requests to the server and load new information whenever it's needed.

For example, we can use a network request to:

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 Asynchronous JavaScript And XML) for network requests from JavaScript. We don't have to use XML though: the term comes from old times, that's why that word is there. You may have heard that term already.

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's not supported by old browsers (can be polyfilled), but very well supported among the modern ones.

The basic syntax is:

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

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

Without options, this is a simple GET request, downloading the contents of the url.
The browser starts the request right away and returns a promise that the calling code should use to get the result.
Getting a response is usually a two-stage process.
First, the promise, returned by fetch, resolves with an object of the built-in Response class as soon as the server responds with headers.
At this stage 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. Abnormal HTTP-statuses, such as 404 or 500 do not cause an error.
We can see HTTP-status in response properties:

status – HTTP status code, e.g. 200.
ok – boolean, true if the HTTP status code is 200-299.

For example:

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

if (response.ok) { // if HTTP-status is 200-299
  // get the response body (the method explained 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.text() – read the response and return as text,
response.json() – parse the response as JSON,
response.formData() – return the response as FormData object (explained in the next chapter),

response.blob() – return the response as Blob (binary data with type),
response.arrayBuffer() – return the response as ArrayBuffer (low-level representation of binary data),
additionally, response.body is a ReadableStream object, it allows you 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 url = 'https://api.github.com/repos/javascript-tutorial/en.javascript.info/commits';
let response = await fetch(url);

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 response 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 a logo image of "fetch" specification (see chapter Blob for details about operations on Blob):

```
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-reading method.
If we've already 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)
```

Response headersThe response headers are available in a Map-like headers object in response.headers.
It's not exactly a Map, but it has similar methods to get individual headers by name or iterate over them:

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

// 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}`);
}
```

Request headersTo set a request header in fetch, we can use the headers option. It has an object with outgoing headers, like this:

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

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

method – HTTP-method, e.g. POST,
body – the request body, one of:

a string (e.g. JSON-encoded),
FormData object, to submit the data as multipart/form-data,
Blob/BufferSource to send binary data,
URLSearchParams, to submit the data in x-www-form-urlencoded encoding, rarely used.

The JSON format is used most of the time.
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 request body is a string, then Content-Type header is set to text/plain;charset=UTF-8 by default.

But, as we're going to send JSON, we use headers option to send application/json instead, the correct Content-Type for JSON-encoded data.

Sending an imageWe can also submit binary data with fetch using Blob or BufferSource objects.

In this example, there's a <canvas> where we can draw by moving a mouse over it. A click on the "submit" button sends the image to the 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
    });

    // the server responds with confirmation and the image size
    let result = await response.json();
    alert(result.message);
  }

</script>
</body>
```

Please note, here we don't set Content-Type header manually, because a Blob object has a built-in type (here image/png, as generated by toBlob). For Blob objects that type becomes the value of Content-Type.
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');
}
```

SummaryA 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, without await:

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

Fetch options so far:

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

In the next chapters we'll see more options and use cases of fetch.
TasksFetch users from GitHubCreate 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:

There should be one fetch request per user.
Requests shouldn't wait for each other. So that the data arrives as soon as possible.
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.solutionTo fetch a user we need: fetch('https://api.github.com/users/USERNAME').
If the response has status 200, call .json() to read the JS object.
Otherwise, 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.Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/fetch","identifier":"\/fetch"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialNetwork requests{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":

[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/
network"}]}June 22, 2021FormDataThis chapter is about sending HTML forms: with or
without files, with additional fields and so on.

FormData objects can help with that. As you might have guessed, it's the object to
represent HTML form data.

The constructor is:

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

If HTML form element is provided, it automatically captures its fields.

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: multipart/form-
data.

From the server point of view, that looks like a usual form submission.

Sending a simple formLet'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>
```

In this example, the server code is not presented, as it's beyond our scope. The server accepts the POST request and replies "User saved".
FormData MethodsWe 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 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 one field with such name, the rest is just like append:

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 fileThe form is always sent as Content-Type: multipart/form-data, 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>
```

Sending a form with Blob dataAs we've seen in the chapter Fetch, it's easy to send dynamically generated binary data e.g. an image, as Blob. 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, as a form, 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>
```

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) with the data imageBlob (2nd argument) from their filesystem.
The server reads form data and the file, as if it were a regular form submission.
SummaryFormData 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 object without a form at all, and then append fields with methods:

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

Let's note two peculiarities here:

The set method removes fields with the same name, append doesn't. That's the only difference between them.
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!
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/formdata","identifier":"\/ formdata"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialNetwork requests{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/ network"}]}June 22, 2021Fetch: Download progressThe 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 ReadableStream – a special object that provides body chunk-by-chunk, as it comes. Readable streams are described in the Streams API specification.
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, otherwise false.
value – a typed array of bytes: Uint8Array.

Please note:
Streams API also describes asynchronous iteration over ReadableStream with for await..of loop, but it's not yet widely supported (see browser issues), so we use while loop.

We receive response chunks in the loop, until the loading finishes, that is: until done becomes true.
To log the progress, we just need for every received fragment value to add its length to the counter.
Here's the full working example that gets the response and logs the progress in console, more explanations to follow:

```
// Step 1: start the fetch and obtain a reader
let response = await fetch('https://api.github.com/repos/javascript-tutorial/
en.javascript.info/commits?per_page=100');

const reader = response.body.getReader();

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

```javascript
// Step 3: read the data
let receivedLength = 0; // received that many bytes 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:

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.

Prior to reading, we can figure out the full response length from the Content-Length header.
It may be absent for cross-origin requests (see chapter Fetch: Cross-Origin Requests) and, well, technically a server doesn't have to set it. But usually it's at place.

Call await reader.read() until it's done.
We gather response chunks in the array chunks. 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).

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:

We create chunksAll = new Uint8Array(receivedLength) – a same-typed array with the combined length.
Then use .set(chunk, position) method to copy each chunk one after another in it.

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, if necessary.
What if we need binary content instead of a string? That's even simpler. Replace steps 4 and 5 with a single line that creates 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.
Also, if the size is unknown, we should check receivedLength in the loop and break it once it reaches a certain limit. So that the chunks won't overflow the memory.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/fetch-progress","identifier":"\/

fetch-progress"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialNetwork requests{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/network"}]}April 13, 2022Fetch: AbortAs we know, fetch returns a promise. And JavaScript generally has no concept of "aborting" a promise. So how can we cancel an ongoing fetch? E.g. if the user actions on our site indicate that the fetch isn't needed any more.

There's a special built-in object for such purposes: AbortController. It can be used to abort not only fetch, but other asynchronous tasks as well.

The usage is very straightforward:

The AbortController objectCreate a controller:

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

A controller is an extremely simple object.

It has a single method abort(),
And a single property signal that allows to set event listeners on it.

When abort() is called:

controller.signal emits the "abort" event.
controller.signal.aborted property becomes true.

Generally, we have two parties in the process:

The one that performs a cancelable operation, it sets a listener on controller.signal.
The one that cancels: it calls controller.abort() when needed.

Here's the full example (without fetch yet):

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

// The party that performs a cancelable operation
// gets the "signal" object
// and sets the listener to trigger when controller.abort() is called
signal.addEventListener('abort', () => alert("abort!"));

// The other party, that cancels (at any point later):
controller.abort(); // abort!

// The event triggers and signal.aborted becomes true
alert(signal.aborted); // true
```

As we can see, AbortController is just a mean to pass abort events when abort() is called on it.
We could implement the same kind of event listening in our code on our own, without the AbortController object.
But what's valuable is that fetch knows how to work with the AbortController object. It's integrated in it.
Using with fetchTo be able to cancel fetch, pass the signal property of an AbortController as a fetch option:

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

The fetch method knows how to work with AbortController. It will listen to abort events on signal.
Now, 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 AbortError, so we should
handle it, e.g. in try..catch.
Here's the full example with fetch aborted after 1 second:

```
// 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 scalableAbortController is scalable. It allows to cancel multiple
fetches at once.
Here's a sketch of code that fetches many urls in parallel, and uses a single controller
to abort them all:

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

```
let controller = new AbortController();

// an array of fetch promises
let fetchJobs = urls.map(url => fetch(url, {
  signal: controller.signal
}));

let results = await Promise.all(fetchJobs);

// if controller.abort() is called from anywhere,
// it aborts all fetches
```

If we have our own asynchronous tasks, different from fetch, we can use a single AbortController to stop those, together with fetches.
We just need to listen to its abort event in our tasks:

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

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

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

// Wait for fetches and our task in parallel
let results = await Promise.all([...fetchJobs, ourJob]);

// if controller.abort() is called from anywhere,
// it aborts all fetches and ourJob
```

Summary
AbortController is a simple object that generates an abort event on its signal property when the abort() method is called (and also sets signal.aborted to true).
fetch integrates with it: we pass the signal property as the option, and then fetch listens

to it, so it's possible to abort the fetch.

We can use AbortController in our code. The "call abort()" !' "listen to abort event" interaction is simple and universal. We can use it even without fetch.

October 18, 2022Fetch: Cross-Origin RequestsIf we send a fetch request to another web-site, it will probably fail.

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.

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.

Why is CORS needed? A brief historyCORS exists to 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. an evil script from website hacker.com could not access the user's mailbox at website 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 and make requests to other websites.

Using formsOne 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, as forms can send data anywhere. But as it's forbidden to access the content of an <iframe> from another site, it wasn't possible to read the response.

To be precise, there were actually tricks for that, they required special scripts at both the iframe and the page. So the communication with the iframe was technically possible. Right now there's no point to go into details, let these dinosaurs rest in peace.

Using scriptsAnother trick was to use a script tag. A script could have any src, with any domain, like <script src="http://another.com/…">. It's possible to execute a script from any website.

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

Here's how it worked.

Let's say we, at our site, need to get the data from http://another.com, such as the weather:

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

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

Then we make a <script> tag with src="http://another.com/weather.json?callback=gotWeather", using the name of our function as the callback URL-parameter.

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

The remote server another.com 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
});
```

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 in browser JavaScript.

At first, cross-origin requests were forbidden. But as a result of long discussions, cross-origin requests were allowed, but with any new capabilities requiring an explicit allowance by the server, expressed in special headers.

Safe requestsThere are two types of cross-origin requests:

Safe requests.
All the others.

Safe Requests are simpler to make, so let's start with them.
A request is safe if it satisfies two conditions:

Safe method: GET, POST or HEAD
Safe 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 "unsafe". 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 safe 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 safe 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 unsafe 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, an unsafe request is not sent.

Now we'll go into details.

CORS for safe requestsIf a request is cross-origin, the browser always adds the Origin header to it.

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

    GET /request

Host: anywhere.com
Origin: https://javascript.info
...

As you can see, the Origin header 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, add 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 it's an error.
The browser plays the role of a trusted mediator here:

It ensures that the correct Origin is sent with a cross-origin request.
It checks for permitting Access-Control-Allow-Origin in the response, if it exists, then JavaScript is allowed to access the response, otherwise it fails 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 headersFor cross-origin request, by default JavaScript may only access so-called "safe" response headers:

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

Accessing any other response header causes an error.

To grant JavaScript access to any other response header, the server must send the Access-Control-Expose-Headers header. It contains a comma-separated list of unsafe header names that should be made accessible.
For example:

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

With such an Access-Control-Expose-Headers header, the script is allowed to read the Content-Encoding and API-Key headers of the response.
"Unsafe" requestsWe can use any HTTP-method: not just GET/POST, but also PATCH, DELETE and others.
Some time ago no one could even imagine that a webpage could make such requests. So there may still 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 "unsafe" request – that couldn't be done in the old times, the browser does not make such requests right away. First, it sends a preliminary, so-called "preflight" request, to ask for permission.
A preflight request uses the method OPTIONS, no body and three headers:

Access-Control-Request-Method header has the method of the unsafe request.
Access-Control-Request-Headers header provides a comma-separated list of its unsafe HTTP-headers.
Origin header tells from where the request came. (such as https://javascript.info)

If the server agrees to serve the requests, then it should respond with empty body, status 200 and headers:

Access-Control-Allow-Origin must be either * or the requesting origin, such as https://javascript.info, to allow it.
Access-Control-Allow-Methods must have the allowed method.
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 the example of a cross-origin 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 unsafe (one is enough):

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

Step 1 (preflight request)Prior to sending such a 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 "unsafe" headers.

Step 2 (preflight response)The server should respond with status 200 and the headers:

Access-Control-Allow-Origin: https://javascript.info
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, it makes sense to allow them in advance by adding them to the list.
For example, this response also allows PUT, DELETE and additional headers:

200 OK
Access-Control-Allow-Origin: https://javascript.info
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 Access-Control-Allow-Methods and Content-Type,API-Key are in the list Access-Control-Allow-Headers, so it sends out the main request.
If there's the header Access-Control-Max-Age with a number of seconds, then the preflight permissions are cached for the given time. The response above will be cached for 86400 seconds (one day). Within this timeframe, subsequent requests will not cause a preflight. Assuming that they fit the cached allowances, they will be sent directly.
Step 3 (actual request)When the preflight is successful, the browser now makes the main request. The process here is the same as for safe requests.
The main request has the 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 main response. A successful preflight does not relieve from that:

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

Then JavaScript is able to read the main server response.

Please note:
Preflight request occurs "behind the scenes", it's invisible to JavaScript. JavaScript only gets the response to the main request or an error if there's no server permission.

CredentialsA cross-origin request initiated by JavaScript code 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. Cross-origin requests made by JavaScript methods on the other hand 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 without them. If allowed, it grants JavaScript the full power to act on behalf of the user and access sensitive information using their credentials.
Does the server really trust the script that much? Then it must explicitly allow requests with credentials with an additional header.
To send credentials in fetch, we need to add the option credentials: "include", like this:

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

Now fetch sends cookies originating from another.com with request to that site. If the server agrees 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. Like shown above, it must provide the exact origin there. That's an additional safety measure, to ensure that the server really knows who it trusts to make such requests.

SummaryFrom the browser point of view, there are two kinds of cross-origin requests: "safe" and all the others.

"Safe" 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 safe requests were doable since ancient times using <form> or <script> tags, while unsafe were impossible for browsers for a long time. So, the practical difference is that safe requests are sent right away, with the Origin header, while for the other ones the browser makes a preliminary "preflight" request, asking for permission.

For safe requests:

!' The browser sends the Origin header with the origin.
!• For requests without credentials (not sent by 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, to grant JavaScript access to any response headers except Cache-Control, Content-Language, Content-Type, Expires, Last-Modified or Pragma, the server should list the allowed ones in Access-Control-Expose-Headers header.
For unsafe requests, a preliminary "preflight" request is issued before the requested one:

!' The browser sends an OPTIONS request to the same URL, with the headers:

Access-Control-Request-Method has requested method.
Access-Control-Request-Headers lists unsafe requested headers.

!• The server should respond with status 200 and the 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 the permissions.

Then the actual request is sent, and the previous "safe" scheme is applied.

TasksWhy do we need Origin?importance: 5As 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:

Why Origin is needed, if Referer has even more information?
Is it possible that there's no Referer or Origin, or is it incorrect?

solutionWe 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 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.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/fetch-crossorigin","identifier":"\/fetch-crossorigin"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialNetwork requests{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/ network"}]}October 12, 2022Fetch APISo far, we know quite a bit about fetch. Let's see the rest of API, to cover all its abilities.

> Please note:
> Please note: most of these options are used rarely. You may skip this chapter and still use fetch well.
Still, it's good to know what fetch can do, so if the need arises, you can return and read the details.

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 current origin
  referrerPolicy: "strict-origin-when-cross-origin", // 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 remaining capabilities.
referrer, referrerPolicyThese options govern how fetch sets the HTTP Referer header.
Usually that header is set automatically and contains the url of the page that made the request. In most scenarios, it's not important at all, 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 remove it.
To send no referrer, 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.
Requests are split into 3 types:

Request to the same origin.
Request to another origin.
Request from HTTPS to HTTP (from safe to unsafe protocol).

Unlike the referrer option that allows to set the exact Referer value, referrerPolicy tells
the browser general rules for each request type.
Possible values are described in the Referrer Policy specification:

"strict-origin-when-cross-origin" – the default value: for same-origin send the full
Referer, for cross-origin send only the origin, unless it's HTTPS!'HTTP request, then
send nothing.
"no-referrer-when-downgrade" – full Referer is always sent, unless we send a request
from HTTPS to HTTP (to the less secure protocol).
"no-referrer" – never send Referer.
"origin" – only send the origin in Referer, not the full page URL, e.g. only http://site.com
instead of http://site.com/path.
"origin-when-cross-origin" – send the full Referer to the same origin, but only the origin
part for cross-origin requests (as above).
"same-origin" – send the full Referer to the same origin, but no Referer for cross-origin
requests.
"strict-origin" – send only the origin, not the Referer for HTTPS!'HTTP requests.
"unsafe-url" – always send the full url in Referer, even for HTTPS!'HTTP requests.

Here's a table with all combinations:

Value
To same origin
To another origin
HTTPS!'HTTP

"no-referrer"

-

-

-

"no-referrer-when-downgrade"

full

full

-

"origin"

origin

origin

origin

"origin-when-cross-origin"

full

origin

origin

"same-origin"

full

-

-

"strict-origin"

origin

origin

-

"strict-origin-when-cross-origin" or "" (default)

full

origin

-

"unsafe-url"

full

full

full

Let's say we have an admin zone with a URL structure that shouldn't be known from outside of the site.
If we send a fetch, then by default it always 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 other websites know only the origin part, not the URL-path, we can set the option:

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

We can put it to all fetch calls, maybe integrate into JavaScript library of our project that does all requests and uses fetch inside.
Its only difference compared to the default behavior is that for requests to another origin fetch sends only the origin part of the URL (e.g. https://javascript.info, without path). For requests to our origin we still get the full Referer (maybe useful for debugging purposes).

Referrer policy is not only for fetch
Referrer policy, described in the specification, is not just for fetch, but more global.
In particular, it's possible to set the default policy for the whole page using the Referrer-Policy HTTP header, or per-link, with <a rel="noreferrer">.

modeThe mode option is a safe-guard that prevents occasional 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 safe cross-origin requests are allowed.

This option may be useful when the URL for fetch comes from a 3rd-party, and we want a "power off switch" to limit cross-origin capabilities.
credentialsThe 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 Access-Control-Allow-Credentials from cross-origin server in order for JavaScript to access the response, that was covered in the chapter Fetch: Cross-Origin Requests,
"omit" – never send, even for same-origin requests.

cacheBy default, fetch requests make use of standard HTTP-caching. That is, it respects the Expires and 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 the cache with the response (if the response headers permit this action),
"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".

redirectNormally, 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" – allows to process HTTP-redirects manually. In case of redirect, we'll get a special response object, with response.type="opaqueredirect" and zeroed/empty status and most other properies.

integrityThe 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 the browser.
For example, we're downloading a file, and we know that its SHA-256 checksum is "abcdef" (a real checksum is longer, of course).
We can put it in the integrity option, like this:


```
fetch('http://site.com/file', {
  integrity: 'sha256-abcdef'
```

```
});
```

Then fetch will calculate SHA-256 on its own and compare it with our string. In case of a mismatch, an error is triggered.
keepaliveThe keepalive option indicates that the request may "outlive" the webpage that initiated it.
For example, we gather statistics on how the current visitor uses our page (mouse clicks, page fragments he views), to analyze and improve the user experience.
When the visitor leaves our page – we'd like to save the data to our server.
We can use the window.onunload event 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 the keepalive option tells the browser to perform the request in the background, even after it leaves the page. So this option is essential for our request to succeed.
It has a few limitations:

We can't send megabytes: the body limit for keepalive requests is 64KB.

If we need to gather a lot of statistics about the visit, we should send it out regularly in packets, so that there won't be a lot left for the last onunload request.
This limit applies to all keepalive requests together. In other words, we can perform multiple keepalive requests in parallel, but the sum of their body lengths should not

exceed 64KB.

We can't handle the server response if the document is unloaded. So in our example fetch will succeed due to keepalive, but subsequent functions won't work.

In most cases, such as sending out statistics, it's not a problem, as the server just accepts the data and usually sends an empty response to such requests.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/fetch-api","identifier":"\/fetch-api"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialNetwork requests{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/ network"}]}December 12, 2021URL objectsThe built-in URL class provides a convenient interface for creating and parsing URLs.
There are no networking methods that require exactly a URL object, strings are good enough. So technically we don't have to use URL. But sometimes it can be really helpful.
Creating a URLThe syntax to create a new URL object:

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

url – the full URL or only path (if base is set, see below),
base – an optional base URL: if set and url argument has only path, then the URL is generated relative to base.

For example:

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

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
```

We can easily create a new URL based on the path relative to an existing URL:

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

alert(newUrl); // 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 for URL components:

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).

We can pass URL objects to networking (and most other) methods instead of a string
We can use a URL object in fetch or XMLHttpRequest, almost everywhere where a URL-string is expected.
Generally, the URL object can be passed to any method instead of a string, as most methods will perform the string conversion, that turns a URL object into a string with full URL.

SearchParams "?…"Let's say we want to create a 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 a URL property for that: url.searchParams, an object of type URLSearchParams.
It provides convenient methods for search parameters:

append(name, value) – add the parameter by name,
delete(name) – remove the parameter by name,
get(name) – get the parameter by name,
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 by name,
set(name, value) – set/replace the parameter,
sort() – sort parameters by name, rarely needed,
…and it's also iterable, similar to Map.

An example with parameters that contain spaces and punctuation marks:

```
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?q=test+me%21

url.searchParams.set('tbs', 'qdr:y'); // added parameter with a colon :
```

```
// parameters are automatically encoded
alert(url); // https://google.com/search?q=test+me%21&tbs=qdr%3Ay

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

EncodingThere's a standard RFC3986 that defines which characters are allowed in URLs 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, but that's an exception).

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

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

let url = new URL('https://ru.wikipedia.org/wiki/B$5D B');

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

As you can see, both B$5D B in the url path and D¢ –â F†R   ameter are encoded.

The URL became longer, because each cyrillic letter is represented with two bytes in UTF-8, so there are two %.. entities.

Encoding stringsIn old times, before URL objects appeared, people used strings for URLs.

As of now, URL objects are often more convenient, but strings can still be used as well.

In many cases using a string makes the code shorter.
If we use a string though, we need to encode/decode special characters manually.
There are built-in functions for that:

encodeURI – encodes URL as a whole.
decodeURI – decodes it back.
encodeURIComponent – encodes a URL component, such as a search parameter, or a hash, or a pathname.
decodeURIComponent – decodes it back.

A natural question is: "What's the difference between encodeURIComponent and encodeURI? When we should use either?"
That's easy to understand if we look at the URL, that's split into components in the picture above:
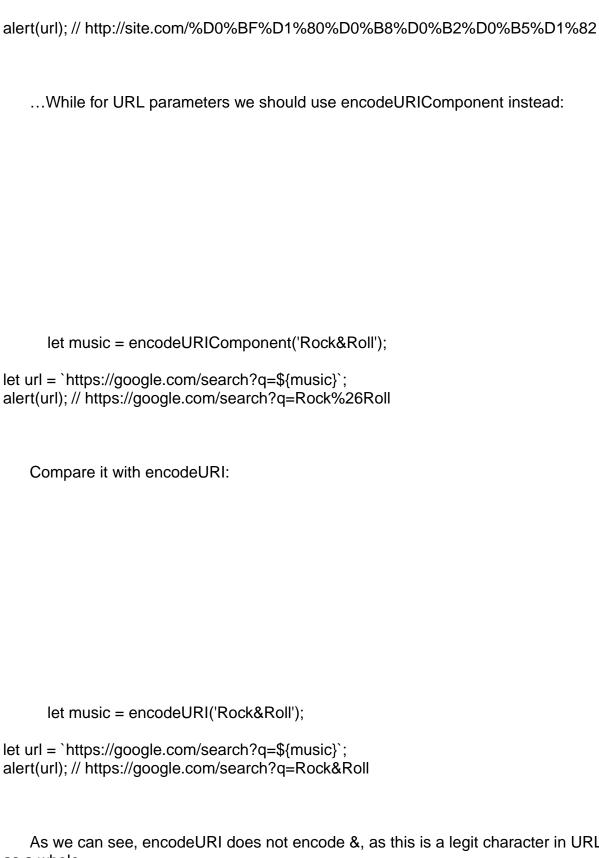
https://site.com:8080/path/page?p1=v1&p2=v2#hash

As we can see, characters such as :, ?, =, &, # are allowed in URL.
…On the other hand, if we look at a single URL component, such as a search parameter, these characters must be encoded, not to break the formatting.

encodeURI encodes only characters that are totally forbidden in URL.
encodeURIComponent encodes same characters, and, in addition to them, characters #, $, &, +, ,, /, :, ;, =, ? and @.

So, for a whole URL we can use encodeURI:

```
// using cyrillic characters in url path
let url = encodeURI('http://site.com/Cô@C„2CTB');
```

alert(url); // http://site.com/%D0%BF%D1%80%D0%B8%D0%B2%D0%B5%D1%82

…While for URL parameters we should use encodeURIComponent instead:

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

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

Compare it with encodeURI:

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

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

As we can see, encodeURI 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 it has only allowed characters.

Encoding difference compared to URL
Classes URL and URLSearchParams are based on the latest URI specification:
RFC3986, while encode* functions are based on the obsolete version RFC2396.
There are a few differences, e.g. IPv6 addresses are encoded differently:

```
// valid url with IPv6 address
let url = 'http://[2607:f8b0:4005:802::1007]/';

alert(encodeURI(url)); // http://%5B2607:f8b0:4005:802::1007%5D/
alert(new URL(url)); // http://[2607:f8b0:4005:802::1007]/
```

As we can see, encodeURI 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.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before
commenting…If you have suggestions what to improve - please submit a GitHub issue
or a pull request instead of commenting.If you can't understand something in the article
– please elaborate.To insert few words of code, use the <code> tag, for several lines –
wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin,
codepen…)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https:\/\/javascript.info\/url","identifier":"\/url"}); };var
disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialNetwork requests{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/

network"}]}May 14, 2022XMLHttpRequestXMLHttpRequest is a built-in browser object that allows to make HTTP requests in JavaScript.

Despite 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 is used for three reasons:

Historical reasons: we need to support existing scripts with XMLHttpRequest.
We need to support old browsers, and don't want polyfills (e.g. to keep scripts tiny).
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 basicsXMLHttpRequest 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:

Create XMLHttpRequest:

```
let xhr = new XMLHttpRequest();
```

The constructor has no arguments.

Initialize it, usually right after new XMLHttpRequest:

```
xhr.open(method, URL, [async, user, password])
```

This method 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.

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 of that later.

Listen to xhr events for response.
These three events are the most widely used:

load – when the request is complete (even if HTTP status is like 400 or 500), and the response is fully downloaded.
error – when the request couldn't be made, e.g. network down or invalid URL.
progress – triggers periodically while the response is being downloaded, reports how much has been 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`); // response is the server response
  }
};

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 xhr properties:

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 body.

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.

URL search parameters
To add parameters to URL, 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 TypeWe 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) or HTML document (based on the MIME type of the received data),
"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 statesXMLHttpRequest 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 requestWe 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 requestsIf 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-headersXMLHttpRequest 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;
 }, {});

// headers['Content-Type'] = 'image/png'
```

POST, FormDataTo 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 <form>
formData.append(name, value); // appends a field
```

We create it, optionally fill from a form, append more fields if needed, and then:

```
xhr.open('POST', ...) – use POST method.
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);
```

```
  xhr.onload = () => alert(xhr.response);
</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 any body, including Blob and BufferSource objects.
Upload progressThe progress event triggers only 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, without methods, exclusively to track upload events: xhr.upload.
It generates events, similar to xhr, but xhr.upload triggers them solely 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 requestsXMLHttpRequest can make cross-origin 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.
SummaryTypical 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 response.
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 properties of the request object xhr 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.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https:\/\/javascript.info\/xmlhttprequest","identifier":"\/

xmlhttprequest"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialNetwork requests{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/
network"}]}October 13, 2022Resumable file uploadWith 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 eventTo 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 won't help us to resume the upload 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 and didn't reach the
receiver.

That's why 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, so we'll make an additional request.

Algorithm

First, create a file id, to uniquely identify the file we're going to upload:

```
let fileId = file.name + '-' + file.size + '-' + file.lastModified;
```

That's needed for resume upload, to tell the server what we're resuming.
If the name or the size or the last modification date changes, then there'll be another
fileId.

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.
If the file doesn't yet exist at the server, then the server response should be 0

Then, we can use Blob method slice to send the file from startByte:

```
xhr.open("POST", "upload");
```

```
// File id, so that the server knows which file we upload
xhr.setRequestHeader('X-File-Id', fileId);
```

```
// 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:
Resultserver.jsuploader.jsindex.htmllet http = require('http');
let static = require('node-static');
let fileServer = new static.Server('.');
let path = require('path');
let fs = require('fs');
let debug = require('debug')('example:resume-upload');

let uploads = Object.create(null);

function onUpload(req, res) {

  let fileId = req.headers['x-file-id'];
  let startByte = +req.headers['x-start-byte'];

  if (!fileId) {
    res.writeHead(400, "No file id");
    res.end();
  }

  // we'll files "nowhere"
  let filePath = '/dev/null';
  // could use a real path instead, e.g.
  // let filePath = path.join('/tmp', fileId);

  debug("onUpload fileId: ", fileId);

  // initialize a new upload
  if (!uploads[fileId]) uploads[fileId] = {};
  let upload = uploads[fileId];

  debug("bytesReceived:" + upload.bytesReceived + " startByte:" + startByte)

  let fileStream;

  // if startByte is 0 or not set, create a new file, otherwise check the size and append to existing one
  if (!startByte) {
    upload.bytesReceived = 0;
    fileStream = fs.createWriteStream(filePath, {
      flags: 'w'
    });
    debug("New file created: " + filePath);
  } else {
    // we can check on-disk file size as well to be sure
    if (upload.bytesReceived != startByte) {

```
      res.writeHead(400, "Wrong start byte");
      res.end(upload.bytesReceived);
      return;
    }
    // append to existing file
    fileStream = fs.createWriteStream(filePath, {
      flags: 'a'
    });
    debug("File reopened: " + filePath);
  }


  req.on('data', function(data) {
    debug("bytes received", upload.bytesReceived);
    upload.bytesReceived += data.length;
  });

  // send request body to file
  req.pipe(fileStream);

  // when the request is finished, and all its data is written
  fileStream.on('close', function() {
    if (upload.bytesReceived == req.headers['x-file-size']) {
      debug("Upload finished");
      delete uploads[fileId];

      // can do something else with the uploaded file here

      res.end("Success " + upload.bytesReceived);
    } else {
      // connection lost, we leave the unfinished file around
      debug("File unfinished, stopped at " + upload.bytesReceived);
      res.end();
    }
  });

  // in case of I/O error - finish the request
  fileStream.on('error', function(err) {
    debug("fileStream error");
    res.writeHead(500, "File error");
    res.end();
  });

}

function onStatus(req, res) {
```

```javascript
  let fileId = req.headers['x-file-id'];
  let upload = uploads[fileId];
  debug("onStatus fileId:", fileId, " upload:", upload);
  if (!upload) {
    res.end("0")
  } else {
    res.end(String(upload.bytesReceived));
  }
}


function accept(req, res) {
  if (req.url == '/status') {
    onStatus(req, res);
  } else if (req.url == '/upload' && req.method == 'POST') {
    onUpload(req, res);
  } else {
    fileServer.serve(req, res);
  }

}



// ----------------------------------

if (!module.parent) {
  http.createServer(accept).listen(8080);
  console.log('Server listening at port 8080');
} else {
  exports.accept = accept;
}class Uploader {

  constructor({file, onProgress}) {
    this.file = file;
    this.onProgress = onProgress;

    // create fileId that uniquely identifies the file
    // we could also add user session identifier (if had one), to make it even more unique
    this.fileId = file.name + '-' + file.size + '-' + file.lastModified;
  }

  async getUploadedBytes() {
    let response = await fetch('status', {
      headers: {
```

```
      'X-File-Id': this.fileId
    }
  });

  if (response.status != 200) {
    throw new Error("Can't get uploaded bytes: " + response.statusText);
  }

  let text = await response.text();

  return +text;
}

async upload() {
  this.startByte = await this.getUploadedBytes();

  let xhr = this.xhr = new XMLHttpRequest();
  xhr.open("POST", "upload", true);

  // send file id, so that the server knows which file to resume
  xhr.setRequestHeader('X-File-Id', this.fileId);
  // send the byte we're resuming from, so the server knows we're resuming
  xhr.setRequestHeader('X-Start-Byte', this.startByte);

  xhr.upload.onprogress = (e) => {
    this.onProgress(this.startByte + e.loaded, this.startByte + e.total);
  };

  console.log("send the file, starting from", this.startByte);
  xhr.send(this.file.slice(this.startByte));

  // return
  //   true if upload was successful,
  //   false if aborted
  // throw in case of an error
  return await new Promise((resolve, reject) => {

    xhr.onload = xhr.onerror = () => {
      console.log("upload end status:" + xhr.status + " text:" + xhr.statusText);

      if (xhr.status == 200) {
        resolve(true);
      } else {
        reject(new Error("Upload failed: " + xhr.statusText));
      }
    };
```

```
      // onabort triggers only when xhr.abort() is called
      xhr.onabort = () => resolve(false);

    });

  }

  stop() {
    if (this.xhr) {
      this.xhr.abort();
    }
  }

}<!DOCTYPE HTML>

<script src="uploader.js"></script>

<form name="upload" method="POST" enctype="multipart/form-data" action="/upload">
  <input type="file" name="myfile">
  <input type="submit" name="submit" value="Upload (Resumes automatically)">
</form>

<button onclick="uploader.stop()">Stop upload</button>


<div id="log">Progress indication</div>

<script>
  function log(html) {
    document.getElementById('log').innerHTML = html;
    console.log(html);
  }

  function onProgress(loaded, total) {
    log("progress " + loaded + ' / ' + total);
  }

  let uploader;

  document.forms.upload.onsubmit = async function(e) {
    e.preventDefault();

    let file = this.elements.myfile.files[0];
    if (!file) return;
```

```
  uploader = new Uploader({file, onProgress});

  try {
    let uploaded = await uploader.upload();

    if (uploaded) {
      log('success');
    } else {
      log('stopped');
    }

  } catch(err) {
    console.error(err);
    log('error');
  }
};
```

</script>As we can see, modern networking methods are close to file managers in their capabilities – control over headers, progress indicator, sending file parts, etc.
We can implement resumable upload and much more.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:∨javascript.info∨resume-upload","identifier":"∨resume-upload"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialNetwork requests{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/ network"}]}December 12, 2022Long pollingLong polling is the simplest way of having persistent connection with server, that doesn't use any specific protocol like WebSocket or Server Sent Events.
Being very easy to implement, it's also good enough in a lot of cases.
Regular PollingThe simplest way to get new information from the server is periodic polling. That is, regular requests to the server: "Hello, I'm here, do you have any information for me?". For example, once every 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:

Messages are passed with a delay up to 10 seconds (between requests).
Even if there are no messages, the server is bombed with requests every 10 seconds, even if the user switched somewhere else or is asleep. That's quite a load to handle,

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 pollingSo-called "long polling" is a much better way to poll the server.
It's also very easy to implement, and delivers messages without delays.
The flow:

A request is sent to the server.
The server doesn't close the connection until it has a message to send.
When a message appears – the server responds to the request with it.
The browser makes a new request immediately.

This situation, where the browser has sent a request and keeps a pending connection with the server, is standard for this method. Only when a message is delivered, the connection is closed and reestablished.

If the connection is lost, because of, say, a network error, the browser immediately sends a new request.
A sketch of client-side subscribe function that makes long requests:

```javascript
async function subscribe() {
  let response = await fetch("/subscribe");

  if (response.status == 502) {
    // Status 502 is a connection timeout error,
    // may happen when the connection was pending for too long,
    // and the remote server or a proxy closed it
    // let's reconnect
    await subscribe();
  } else if (response.status != 200) {
    // An error - let's show it
    showMessage(response.statusText);
    // Reconnect in one second
    await new Promise(resolve => setTimeout(resolve, 1000));
    await subscribe();
  } else {
    // Get and show the message
    let message = await response.text();
```

```
    showMessage(message);
    // Call subscribe() again to get the next message
    await subscribe();
  }
}


subscribe();
```

As you can see, 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 one process per connection, resulting in there being as many processes as there are connections, while each process consumes quite a bit of memory. So, too many connections will just consume it all.
That's often the case for backends written in languages like PHP and Ruby.
Servers written using Node.js usually don't have such problems.
That said, it isn't a programming language issue. Most modern languages, including PHP and Ruby allow to implement a proper backend. Just please make sure that your server architecture works fine with many simultaneous connections.

Demo: a chatHere's a demo chat, you can also download it and run locally (if you're familiar with Node.js and can install modules):
Resultbrowser.jsserver.jsindex.html// Sending messages, a simple POST
function PublishForm(form, url) {

```
  function sendMessage(message) {
    fetch(url, {
      method: 'POST',
      body: message
    });
  }

  form.onsubmit = function() {
    let message = form.message.value;
    if (message) {
      form.message.value = '';
      sendMessage(message);
    }
    return false;
  };
}
```

```javascript
// Receiving messages with long polling
function SubscribePane(elem, url) {

  function showMessage(message) {
    let messageElem = document.createElement('div');
    messageElem.append(message);
    elem.append(messageElem);
  }

  async function subscribe() {
    let response = await fetch(url);

    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();

}let http = require('http');
let url = require('url');
let querystring = require('querystring');
let static = require('node-static');

let fileServer = new static.Server('.');

let subscribers = Object.create(null);

function onSubscribe(req, res) {
  let id = Math.random();

  res.setHeader('Content-Type', 'text/plain;charset=utf-8');
```

```
    res.setHeader("Cache-Control", "no-cache, must-revalidate");

  subscribers[id] = res;

  req.on('close', function() {
    delete subscribers[id];
  });

}

function publish(message) {

  for (let id in subscribers) {
    let res = subscribers[id];
    res.end(message);
  }

  subscribers = Object.create(null);
}

function accept(req, res) {
  let urlParsed = url.parse(req.url, true);

  // new client wants messages
  if (urlParsed.pathname == '/subscribe') {
    onSubscribe(req, res);
    return;
  }

  // sending a message
  if (urlParsed.pathname == '/publish' && req.method == 'POST') {
    // accept POST
    req.setEncoding('utf8');
    let message = '';
    req.on('data', function(chunk) {
      message += chunk;
    }).on('end', function() {
      publish(message); // publish it to everyone
      res.end("ok");
    });

    return;
  }

  // the rest is static
  fileServer.serve(req, res);
```

```
  }

  function close() {
    for (let id in subscribers) {
      let res = subscribers[id];
      res.end();
    }
  }

  // ----------------------------------

  if (!module.parent) {
    http.createServer(accept).listen(8080);
    console.log('Server running on port 8080');
  } else {
    exports.accept = accept;

    if (process.send) {
      process.on('message', (msg) => {
        if (msg === 'shutdown') {
          close();
        }
      });
    }

    process.on('SIGINT', close);
  }
```

```html
<!DOCTYPE html>
<script src="browser.js"></script>
```

All visitors of this page will see messages of each other.

```html
<form name="publish">
  <input type="text" name="message" />
  <input type="submit" value="Send" />
</form>

<div id="subscribe">
</div>

<script>
  new PublishForm(document.forms.publish, 'publish');
  // random url parameter to avoid any caching issues
  new SubscribePane(document.getElementById('subscribe'), 'subscribe?random=' +
Math.random());
</script>
```

Browser code is in browser.js.

Area of usageLong 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.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/long-polling","identifier":"\/long-polling"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialNetwork requests{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/network"}]}October 14, 2022WebSocketThe WebSocket protocol, described in the specification RFC 6455, provides a way to exchange data between browser and server via a persistent connection. The data can be passed in both directions as "packets", without breaking the connection and the need of additional HTTP-requests.

WebSocket is especially great for services that require continuous data exchange, e.g. online games, real-time trading systems and so on.

A simple exampleTo 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 is 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 the sender and decrypts it at the receiver. So data packets are passed encrypted through proxies. They can't see what's inside and let them 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:

```javascript
let socket = new WebSocket("wss://javascript.info/article/websocket/demo/hello");

socket.onopen = function(e) {
  alert("[open] Connection established");
  alert("Sending to server");
  socket.send("My name is John");
};

socket.onmessage = function(event) {
  alert(`[message] Data received from server: ${event.data}`);
};

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]`);
};
```

For demo purposes, there's a small server server.js written in Node.js, for the example above, running. It responds with "Hello from server, John", 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 websocketWhen 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 headers for a request made by 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 the 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, used to ensure that the server

supports WebSocket protocol. It's random to prevent proxies from caching any following communication.
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: hsBlbuDTkk24srzEOTBUlZAlC2g=

Here Sec-WebSocket-Accept is Sec-WebSocket-Key, recoded using a special algorithm. Upon seeing it, the browser understands that the server really does support the WebSocket protocol.
Afterwards, the data is transferred using the WebSocket protocol, we'll see its structure ("frames") soon. And that's not HTTP at all.
Extensions and subprotocolsThere 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, functionality that extends the WebSocket protocol. The header Sec-WebSocket-Extensions is sent automatically by the browser, with the list of all extensions it supports.

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. So, this header describes the data formats that we're going to use.
This optional header is set using the second parameter of new WebSocket. That's the array of subprotocols, e.g. if we'd like to use SOAP or WAMP:

```
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: hsBlbuDTkk24srzEOTBUlZAlC2g=
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.
Data transferWebSocket 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.
there's also "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 are required: just send it out in any format.
When we receive the data, text always comes as string. And for binary data, we can choose between Blob and ArrayBuffer formats.
That's set by socket.binaryType property, it's "blob" by default, so binary data comes as Blob objects.
Blob is a high-level binary object, it directly integrates with <a>, <img> 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.binaryType = "arraybuffer";
socket.onmessage = (event) => {
  // event.data is either a string (if text) or arraybuffer (if binary)
};
```

Rate limitingImagine, our app is generating a lot of data to send. But the user has a slow network connection, maybe on a mobile internet, 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 remain 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 closeNormally, 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 the 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 set 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.

The full list can be found in RFC6455, §7.4.1.
WebSocket codes are somewhat like HTTP codes, but different. In particular, codes lower 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 stateTo 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 exampleLet's review a chat example using browser WebSocket API and Node.js WebSocket module https://github.com/websockets/ws. We'll pay the main attention to the client side, but the server is also simple.
HTML: we need 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>
```

From JavaScript we want three things:

Open the connection.
On form submission – socket.send(message) for the message.
On incoming message – append it to div#messages.

Here's the code:

```
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;
};

// message received - show the 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'll use Node.js, but you don't have to. Other platforms also have their means to work with WebSocket.
The server-side algorithm will be:

Create clients = new Set() – a set of sockets.
For each accepted websocket, add it to the set clients.add(socket) and set message event listener to get its messages.
When a message is received: iterate over clients and send it to everyone.
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 here 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:

You can also download it (upper-right button in the iframe) and run it locally. Just don't forget to install Node.js and npm install ws before running.
SummaryWebSocket 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 projects, people run a 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 the WebSocket server, while https://site.com goes to the main HTTP-server.

Surely, other ways of integration are also possible.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/websocket","identifier":"\/ websocket"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialNetwork requests{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Network requests","item":"https://javascript.info/ network"}]}November 30, 2020Server Sent EventsThe 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 messagesTo 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 colon is optional.
Messages are delimited with double line breaks \n\n.
To send a line break \n, we can immediately send one more data: (3rd message above).

In practice, complex messages are usually sent JSON-encoded. Line-breaks are
encoded as \n within them, so multiline data: messages are not necessary.
For instance:

```
data: {"user":"John","message":"First line\n Second 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-origin requestsEventSource supports cross-origin requests, like fetch and 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-origin headers.

ReconnectionUpon 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 many milliseconds before reconnecting. Or longer, e.g. if the browser knows (from OS) that there's no network connection at the moment, it may wait until the connection appears, 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. In such cases the "error" event will be emitted, and the browser won't reconnect.

Please note:
When a connection is finally closed, there's no way to "reopen" it. If we'd like to connect again, just create a new EventSource.

Message idWhen 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.

        Put id: after data:
        Please note: the id is appended below message data by the server, to ensure that lastEventId is updated after the message is received.

Connection status: readyStateThe 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 typesBy 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 exampleHere's the server that sends messages with 1, 2, 3, then bye and breaks the connection.
Then the browser automatically reconnects.
Resultserver.jsindex.htmllet http = require('http');
let url = require('url');
let querystring = require('querystring');

```javascript
let static = require('node-static');
let fileServer = new static.Server('.');

function onDigits(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/event-stream; charset=utf-8',
    'Cache-Control': 'no-cache'
  });

  let i = 0;

  let timer = setInterval(write, 1000);
  write();

  function write() {
    i++;

    if (i == 4) {
      res.write('event: bye\ndata: bye-bye\n\n');
      clearInterval(timer);
      res.end();
      return;
    }

    res.write('data: ' + i + '\n\n');

  }
}

function accept(req, res) {

  if (req.url == '/digits') {
    onDigits(req, res);
    return;
  }

  fileServer.serve(req, res);
}


if (!module.parent) {
  http.createServer(accept).listen(8080);
} else {
  exports.accept = accept;
}<!DOCTYPE html>
<script>
```

```javascript
let eventSource;

function start() { // when "Start" button pressed
  if (!window.EventSource) {
    // IE or an old browser
    alert("The browser doesn't support EventSource.");
    return;
  }

  eventSource = new EventSource('digits');

  eventSource.onopen = function(e) {
    log("Event: open");
  };

  eventSource.onerror = function(e) {
    log("Event: error");
    if (this.readyState == EventSource.CONNECTING) {
      log(`Reconnecting (readyState=${this.readyState})...`);
    } else {
      log("Error has occured.");
    }
  };

  eventSource.addEventListener('bye', function(e) {
    log("Event: bye, data: " + e.data);
  });

  eventSource.onmessage = function(e) {
    log("Event: message, data: " + e.data);
  };
}

function stop() { // when "Stop" button pressed
  eventSource.close();
  log("eventSource.close()");
}

function log(msg) {
  logElem.innerHTML += msg + "<br>";
  document.documentElement.scrollTop = 99999999;
}
</script>

<button onclick="start()">Start</button> Press the "Start" to begin.
<div id="logElem" style="margin: 6px 0"></div>
```

```html
<button onclick="stop()">Stop</button>
```
"Stop" to finish.SummaryEventSource object automatically 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 received identifier is sent in Last-Event-ID header upon reconnection.
The current state is in the readyState property.

That makes EventSource a viable alternative to WebSocket, as the latter is more low-level and lacks such built-in features (though they can be implemented).
In many real-life applications, the power of EventSource is just enough.
Supported in all modern browsers (not IE).
The syntax is:

```javascript
let source = new EventSource(url, [credentials]);
```

The second argument has only one possible option: { withCredentials: true }, it allows sending cross-origin credentials.
Overall cross-origin 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.

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 formatThe server sends messages, delimited by \n\n.
A message may have following fields:

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: – event name, must precede data:.

A message may include one or more fields in any order, but id: usually goes the last.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/server-sent-events","identifier":"\/server-sent-events"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
Tutorial{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}]}
Storing data in the browserCookies, document.cookieLocalStorage, sessionStorageIndexedDBCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial map
TutorialStoring data in the browser{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Storing data in the browser","item":"https://javascript.info/data-storage"}]}October 1, 2022Cookies, document.cookieCookies are small strings of data that are stored directly in the browser. They are a part of the HTTP protocol, defined by the RFC 6265 specification.
Cookies are usually set by a web-server using the response Set-Cookie HTTP-header. Then, the browser automatically adds them to (almost) every request to the same domain using the Cookie HTTP-header.
One of the most widespread use cases is authentication:

Upon sign in, the server uses the Set-Cookie HTTP-header in the response to set a cookie with a unique "session identifier".
Next time when the request is sent to the same domain, the browser sends the cookie over the net using the Cookie HTTP-header.
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.cookieDoes your browser store any cookies from this site? Let's see:

```
// 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.cookieWe can write to document.cookie. But it's not a data property, it's an accessor (getter/setter). An assignment to it is treated specially.
A write operation to document.cookie updates only 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 the document.cookie= operation does not overwrite all cookies. It only sets the mentioned cookie user.

Technically, name and value can have any characters. To keep the valid formatting, they should be escaped using a built-in encodeURIComponent function:

```
    // special characters (spaces), 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 the 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 must be absolute. It makes the cookie accessible for pages under that path. 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 defines where the cookie is accessible. In practice though, there are limitations. We can't set any domain.
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.
By default, a cookie is accessible only at the domain that set it.
Please note, by default a cookie is also not shared to a subdomain as well, such as forum.site.com.

```
// if we set a cookie at site.com website...
document.cookie = "user=John"

// ...we won't see it at forum.site.com
alert(document.cookie); // no user
```

…But this can be changed. If we'd like to allow subdomains like forum.site.com to get a cookie set at site.com, that's possible.

For that to happen, when setting a cookie at site.com, we should explicitly set the domain option to the root domain: domain=site.com. Then all subdomains will see such cookie.

For example:

```
    // 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 (with a dot before site.com) also works the same way, allowing access to the cookie from subdomains. That's an old notation and should be used if we need to support very old browsers.

To summarize, the domain option allows to make a cookie accessible at subdomains.

expires, max-ageBy 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 a browser close, we can set either the expires or max-age option.

expires=Tue, 19 Jan 2038 03:14:07 GMT

The cookie expiration date defines the time, when the browser will automatically delete it.

The date must be exactly in this format, in the 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

It's an alternative to expires and specifies the cookie's expiration in seconds from the current moment.
If set to zero or a negative value, the cookie is deleted:

```
    // cookie will die in +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, the secure flag is the right thing.

```
    // assuming we're on https:// now
// set the cookie to be secure (only accessible over HTTPS)
document.cookie = "user=John; secure";
```

samesiteThat'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 attackImagine, 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 accidentally come to another site evil.com. That site has JavaScript code that submits a form <form action="https://bank.com/pay"> to bank.com with fields that initiate a transaction to the hacker's account.

The browser sends cookies every time you visit the site bank.com, even if the form was submitted from evil.com. So the bank recognizes you and actually performs the payment.

That's a so-called "Cross-Site Request Forgery" (in short, XSRF) attack.
Real banks are protected from it of course. All forms generated by bank.com have a special field, a so-called "XSRF protection token", that an evil page can't generate or extract from a remote page. It can submit a form there, but can't get the data back. The site bank.com checks for such token in every form it receives.

Such a protection takes time to implement though. We need to ensure that every form has the required token field, and we must also check all requests.

Enter cookie samesite optionThe 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 same 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 the samesite option, then a 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, e.g. a form submission from another page at bank.com.

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's website, for the second cookie to be sent.

samesite=lax

A more relaxed approach that also protects from XSRF and doesn't break the 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:

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.

The operation performs a 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, JavaScript methods for network requests do not perform any navigation, hence they don't fit.

So, what samesite=lax does, is to basically allow the most common "go to URL" operation to have cookies. E.g. opening a website link from notes that satisfy these conditions.
But anything more complicated, like a network 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.
There's a drawback:

samesite is ignored (not supported) by very 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.
httpOnlyThis option has nothing to do with JavaScript, but we have to mention it for completeness.
The web-server uses the Set-Cookie header to set a cookie. Also, it may set the httpOnly option.
This option forbids any JavaScript access to the cookie. We can't see such a 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, hackers should not be able to inject their code into our site, but there may be bugs that let them do it.

Normally, if such a thing happens, and a user visits a web-page with hacker's JavaScript 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 functionsHere'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 a 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's name to the given value with path=/ by default (can be modified to add other defaults):

```
     function setCookie(name, value, options = {}) {
```

```javascript
  options = {
    path: '/',
    // add other defaults here if necessary
    ...options
  };

  if (options.expires instanceof Date) {
    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:

```javascript
    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 cookiesA cookie is called "third-party" if it's placed by a domain other than the page the user is visiting.
For instance:

A page at site.com loads a banner from another site: <img src="https://ads.com/banner.png">.

Along with the banner, the remote server at ads.com may set the Set-Cookie header with a cookie like id=1234. Such a cookie originates from the ads.com domain, and will only be visible at ads.com:

Next time when ads.com is accessed, the remote server gets the id cookie and recognizes the user:

What's even more important is, when the user moves from site.com to another site other.com, which 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.

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 – the cookie belongs to the domain of the current webpage.

Appendix: GDPRThis 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 the users' privacy. One of these rules is to require an explicit permission for tracking cookies from the user.
Please note, that's only about tracking/identifying/authorizing 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:

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" (that describes how cookies are used), the user must check it, and then the website is free to set auth cookies.

If a website wants to set tracking cookies for everyone.
To do so legally, a website shows a modal "splash screen" for newcomers, and requires them to agree to the 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 such "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.
Summarydocument.cookie provides access to cookies.

Write operations modify only cookies mentioned in it.

Name/value must be encoded.
One cookie may not exceed 4KB in size. The number of cookies allowed on a domain is around 20+ (varies by 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 the current domain only. If the domain is set explicitly, the cookie becomes visible on subdomains.
expires or max-age sets the 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. This 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.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/cookie","identifier":"\/ cookie"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialStoring data in the browser{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Storing data in the browser","item":"https:// javascript.info/data-storage"}]}October 5, 2022LocalStorage, sessionStorageWeb 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 modern browsers allow at least 5 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 the 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 allows access by index with key(index).
Let's see how it works.
localStorage demoThe 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 accessWe 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, because:

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
```

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 keysAs 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 onlyPlease note that both key and value must be strings.
If they were any other type, like a number, or an object, they would get converted to a string automatically:

```
    localStorage.user = {name: "John"};
alert(localStorage.user); // [object Object]
```
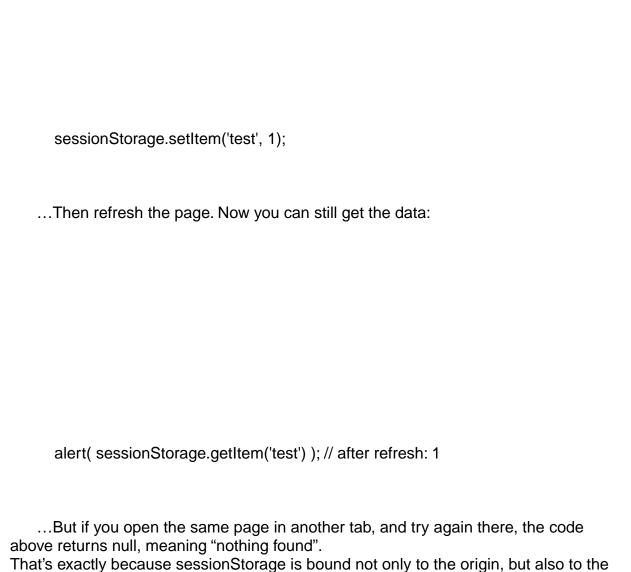
We can use JSON to store objects though:

```
localStorage.user = JSON.stringify({name: "John"});

// sometime later
let user = JSON.parse( localStorage.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) );
```

sessionStorageThe 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 same 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 eventWhen 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.
You might want to open this page in two browser windows to test the code below.
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 => { // can also use window.addEventListener('storage',
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 event.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.
SummaryWeb storage objects localStorage and sessionStorage allow to store key/value pairs in the browser.

Both key and value must be strings.
The limit is 5mb+, 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 (key/oldValue/newValue), the document url and the storage object storageArea.
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).

TasksAutosave a form fieldCreate a textarea field that "autosaves" its value on every change.
So, if the user accidentally closes the page, and opens it again, he'll find his unfinished input at place.
Like this:

Open a sandbox for the task.solutionOpen the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:VVjavascript.infoVlocalstorage","identifier":"Vlocalstorage"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialStoring data in the browser{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Storing data in the browser","item":"https://javascript.info/data-storage"}]}June 20, 2022IndexedDBIndexedDB is a database that is built into a browser, much more powerful than localStorage.

Stores almost any kind of values by keys, multiple key types.
Supports transactions for reliability.
Supports key range queries, indexes.
Can store much bigger volumes of 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 an understanding of IndexedDb, we'll use the wrapper.

Where's the data?
Technically, the data is usually stored in the visitor's home directory, along with browser settings, extensions, etc.
Different browsers and OS-level users have each their own independant storage.

Open databaseTo start working with IndexedDB, we first need to open (connect to) 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 each other's databases.
The call returns openRequest object, we should listen to events on it:

success: database is ready, there's the "database object" in openRequest.result, 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 full-time access to it. So, when we have published a new version of our app, and the user visits our webpage, 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 upgradeneeded event also triggers when the database doesn't yet exist (technically, its version is 0), so we can perform the initialization.
Let's say we published the first version of our app.
Then we can open the database with version 1 and perform the initialization in an upgradeneeded handler like this:

```
    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 working with database using db object
};
```

Then, later, we publish the 2nd version.
We can open it with version 2 and perform the upgrade like this:

```
    let openRequest = indexedDB.open("store", 2);

openRequest.onupgradeneeded = function(event) {
  // the existing database version is less than 2 (or it doesn't exist)
  let db = openRequest.result;
  switch(event.oldVersion) { // existing db version
    case 0:
      // version 0 means that the client had no database
      // perform initialization
    case 1:
      // client had version 1
      // update
  }
};
```

Please note: as our current version is 2, the onupgradeneeded handler has a code branch for version 0, suitable for users that are accessing for the first time and have no database, and also for version 1, for upgrades.
And then, only if onupgradeneeded handler finishes without errors, openRequest.onsuccess triggers, and the database is considered successfully opened.
To delete a database:

```
    let deleteRequest = indexedDB.deleteDatabase(name)
// deleteRequest.onsuccess/onerror tracks the result
```

We can't open a database using an older open call version
If the current user database has a higher version than in the open call, e.g. the

existing DB version is 3, and we try to open(...2), then that's an error, openRequest.onerror triggers.

That's rare, but such a thing may happen when a visitor loads outdated JavaScript code, e.g. from a proxy cache. So the code is old, but his database is new.

To protect from errors, we should check db.version and suggest a page reload. Use proper HTTP caching headers to avoid loading the old code, so that you'll never have such problems.

Parallel update problemAs 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, so our code is newer.

And then the same visitor opens our site in another tab.

So there's a tab with an open connection to DB version 1, while the second one attempts to update it to version 2 in its upgradeneeded handler.

The problem is that a database is shared between two tabs, as it'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, including the one in the first tab.

In order to organize that, the versionchange event triggers on the "outdated" database object. We should listen for it and close the old database connection (and probably suggest a page reload, to load the updated code).

If we don't listen for the versionchange event and don't close the old connection, then the second, new connection won't be made. The openRequest object will emit the blocked event instead of success. So the second tab won't work.

Here's the code to correctly handle the parallel upgrade. It installs the onversionchange handler, that triggers if the current database connection becomes outdated (db version is updated elsewhere) and closes the connection.

```
    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() {
  // this event shouldn't trigger if we handle onversionchange correctly

  // it means that there's another open connection to the same database
  // and it wasn't closed after db.onversionchange triggered for it
};
```

…In other words, here we do two things:

The db.onversionchange listener informs us about a parallel update attempt, if the current database version becomes outdated.
The openRequest.onblocked listener informs us about the opposite situation: there's a connection to an outdated version elsewhere, and it doesn't close, so the newer connection can't be made.

We can handle things more gracefully in db.onversionchange, prompt the visitor to save the data before the connection is closed and so on.
Or, an alternative approach would be to not close the database in db.onversionchange, but instead use the onblocked handler (in the new tab) to alert the visitor, tell him that the newer version can't be loaded until they close other tabs.
These update collisions happen rarely, but we should at least have some handling for them, at least an onblocked handler, to prevent our script from dying silently.
Object storeTo 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 an 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 be one of these types – number, date, string, binary, or array. It's a unique identifier, so 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 setting up an object property as the key, which is 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 only be created/removed/altered during a version update.

To perform a database version upgrade, there are two main approaches:

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).

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 existance. 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')
```

TransactionsThe term "transaction" is generic, used in many kinds of databases. A transaction is a group of operations, that should either all succeed or all fail.
For instance, when a person buys something, we need to:

Subtract the money from their account.
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 upgradeneeded handler. That's why it's a single place where we can update the database structure, create/remove object stores.

Why are there 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 the same store concurrently, 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:

Create a transaction, mentioning all the stores it's going to access, at (1).
Get the store object using transaction.objectStore(name), at (2).
Perform the request to the object store books.add(book), at (3).
…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 the 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' autocommitIn 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 succeed or all fail. How do we mark the transaction as finished, with 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 a transaction. IndexedDB will not keep the transaction waiting till these are done.

In the code below, request2 in the line (*) fails, because the transaction is already committed, and 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 the 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 handlingWrite 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 delegationDo 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
  }
};
```

SearchingThere are two main types of search in an object store:

By a key value or a key range. In our "books" storage that would be a value or range of values of book.id.
By another object field, e.g. book.price. This required an additional data structure, named "index".

By keyFirst let's deal with the first type of search: by key.
Searching methods support both exact key values and so-called "ranges of values" – IDBKeyRange objects that specify an acceptable "key range".
IDBKeyRange objects are created using following calls:

IDBKeyRange.lowerBound(lower, [open]) means: "elower (or >lower if open is true)
IDBKeyRange.upperBound(upper, [open]) means: "dupper (or <upper if open is true)
IDBKeyRange.bound(lower, upper, [lowerOpen], [upperOpen]) means: between lower and upper. If the open flags is true, the corresponding key is not included in the range.
IDBKeyRange.only(key) – a range that consists of only one key, rarely used.

We'll see practical examples of using them very soon.
To perform the actual search, there are following methods. They 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 id < 'html'
books.getAll(IDBKeyRange.upperBound('html', true))

// get all books
books.getAll()

// get all keys, where id > 'js'
books.getAllKeys(IDBKeyRange.lowerBound('js', true))
```

Object store is always sorted
An object store sorts values by key internally.
So requests that return many values always return them in sorted by key order.

By a field using an indexTo 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 = books.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 storeThe 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.
```

CursorsMethods 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 do 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 for the object key:

```
let request = priceIdx.openCursor(IDBKeyRange.upperBound(5));

// called for each record
request.onsuccess = function() {
  let cursor = request.result;
  if (cursor) {
    let primaryKey = 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 wrapperAdding 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 handlingIf 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" pitfallAs 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 the same as when working with native IndexedDB: either make a new transaction or just split things apart.

Prepare the data and fetch all that's needed first.
Then save in the database.

Getting native objectsInternally, 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
```

SummaryIndexedDB 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:

Get a promise wrapper like idb.
Open a database: idb.openDb(name, version, onupgradeneeded)

Create object storages and indexes in onupgradeneeded handler or perform version update if needed.

For requests:

Create transaction db.transaction('books') (readwrite if needed).
Get the object store transaction.objectStore('books').

Then, to search by a key, call methods on the object store directly.

To search by an object field, create an index.

If the data does not fit in memory, use a cursor.

Here's a small demo app:
Resultindex.html<!doctype html>
<script src="https://cdn.jsdelivr.net/npm/idb@3.0.2/build/idb.min.js"></script>

<button onclick="addBook()">Add a book</button>
<button onclick="clearBooks()">Clear books</button>

<p>Books list:</p>

<ul id="listElem">

<script>
let db;

init();

async function init() {
  db = await idb.openDb('booksDb', 1, db => {
    db.createObjectStore('books', {keyPath: 'name'});
  });

  list();
}

async function list() {
  let tx = db.transaction('books');
  let bookStore = tx.objectStore('books');

  let books = await bookStore.getAll();

  if (books.length) {

```
    listElem.innerHTML = books.map(book => `<li>
       name: ${book.name}, price: ${book.price}
     </li>`).join('');
  } else {
    listElem.innerHTML = '<li>No books yet. Please add books.</li>'
  }


}

async function clearBooks() {
  let tx = db.transaction('books', 'readwrite');
  await tx.objectStore('books').clear();
  await list();
}

async function addBook() {
  let name = prompt("Book name?");
  let price = +prompt("Book price?");

  let tx = db.transaction('books', 'readwrite');

  try {
    await tx.objectStore('books').add({name, price});
    await list();
  } catch(err) {
    if (err.name == 'ConstraintError') {
      alert("Such book exists already");
      await addBook();
    } else {
      throw err;
    }
  }
}

window.addEventListener('unhandledrejection', event => {
  alert("Error: " + event.reason.message);
});

</script>
```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:VVjavascript.infoVindexeddb","identifier":"V

indexeddb"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
Tutorial{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}]}
AnimationCSS and JavaScript animations.
Bezier curveCSS-animationsJavaScript animationsCtrl + !•Previous lessonCtrl + !'Next
lessonShareTutorial map
TutorialAnimation{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Animation","item":"https://javascript.info/
animation"}]}November 30, 2022Bezier curveBezier 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.

Some theory, please
This article provides a theoretical, but very needed insight into what Bezier
curves are, while the next one shows how we can use them for CSS animations.
Please take your time to read and understand the concept, it'll serve you well.

Control pointsA 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:

Points are not always on curve. That's perfectly normal, later we'll see how the curve is
built.

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.

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 algorithmThere's a mathematical formula for Bezier curves, but let's cover it a bit later, because
De Casteljau's algorithm 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:

Draw control points. In the demo above they are labeled: 1, 2, 3.

Build segments between control points 1 !' 2 !' 3. In the demo above they are brown.

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 !' 2, 2 !' 3, 3 !' 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:

We connect them to get initially N-1 segments. 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. Repeat step 2 until there is only one point.

These points make the curve. Run and pause examples to clearly see the segments and how the curve is built. 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):

If there's something unclear in the algorithm description, please look at the live examples above to see how the curve is built. 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.

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.

MathsA 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 Pi: the first control point has coordinates P1 = (x1, y1), the second: P2 = (x2, y2), 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)P1 + tP2

For 3 control points:
P = (1" t)2P1 + 2(1" t)tP2 + t2P3

For 4 control points:
P = (1" t)3P1 + 3(1" t)2tP2 +3(1" t)t2P3 + t3P4

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)2x1 + 2(1" t)tx2 + t2x3
y = (1" t)2y1 + 2(1" t)ty2 + t2y3

Instead of x1, y1, x2, y2, x3, y3 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 + t2 * 1 = (1-t)t + t2 = t
y = (1" t)2 * 0 + 2(1" t)t * 1 + t2 * 0 = 2(1-t)t = −2t2 + 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.
SummaryBezier curves are defined by their control points.
We saw two definitions of Bezier curves:

Using a drawing process: De Casteljau's algorithm.
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-animationsCSS animations make it possible to do simple animations without JavaScript at all.
JavaScript can be used to control CSS animations and make them even better, with little code.
CSS transitionsThe 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 fluid transition will be done 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>
```

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 declaring them together in the order: property duration timing-function delay, as well as animating multiple properties at once.
For instance, this button animates both color and font-size:

```html
<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>
```

Now, let's cover animation properties one by one.
transition-propertyIn transition-property, we write a list of properties to animate, for instance: left, margin-left, height, color. Or we could write all, which means "animate all properties".
Do note that, there are properties which can not be animated. However, most of the generally used properties are animatable.

transition-durationIn 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-delayIn transition-delay we can specify the delay before the animation. For instance, if transition-delay is 1s and transition-duration is 2s, then the animation starts 1 second after the property change and the total duration will be 2 seconds.
Negative values are also possible. Then the animation is shown immediately, but the starting point of the animation will be after given value (time). For example, if transition-delay is -1s and transition-duration is 2s, then animation starts from the halfway point and total duration will be 1 second.
Here the animation shifts numbers from 0 to 9 using CSS translate property:
Resultscript.jsstyle.cssindex.htmlstripe.onclick = function() {

```
  stripe.classList.add('animate');
};#digit {
  width: .5em;
  overflow: hidden;
  font: 32px monospace;
  cursor: pointer;
}

#stripe {
  display: inline-block
}

#stripe.animate {
  transform: translate(-90%);
  transition-property: transform;
  transition-duration: 9s;
  transition-timing-function: linear;
}<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  Click below to animate:

  <div id="digit"><div id="stripe">0123456789</div></div>

  <script src="script.js"></script>
</body>

</html>
```

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 could also start it from somewhere in the middle of the transition, from an exact number, e.g. corresponding to the current second, using a negative transition-delay. Here if you click the digit – it starts the animation from the current second: Resultscript.jsstyle.cssindex.htmlstripe.onclick = function() {
```
  let sec = new Date().getSeconds() % 10;
  stripe.style.transitionDelay = '-' + sec + 's';
  stripe.classList.add('animate');
};#digit {
  width: .5em;
  overflow: hidden;
  font: 32px monospace;
  cursor: pointer;
}

#stripe {
  display: inline-block
}

#stripe.animate {
  transform: translate(-90%);
  transition-property: transform;
  transition-duration: 9s;
  transition-timing-function: linear;
}<!doctype html>
```

```
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  Click below to animate:
  <div id="digit"><div id="stripe">0123456789</div></div>

  <script src="script.js"></script>
</body>
</html>
```
JavaScript does it with 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-functionThe timing function describes how the animation process is distributed along its timeline. Will it start slowly and then go fast, or vice versa.
It appears to be the most complicated property at first. 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 with the curve, as it's used more often.
Bezier curveThe timing function can be set as a Bezier curve with 4 control points that satisfy the conditions:

First control point: (0,0).
Last control point: (1,1).
For intermediate points, the 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.

The x axis is the time: 0 – the start, 1 – the end 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):
Resultstyle.cssindex.html.train {

```
.train {
  position: relative;
  cursor: pointer;
  width: 177px;
  height: 160px;
  left: 0;
  transition: left 5s cubic-bezier(0, 0, 1, 1);
}
```

```html
<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  <img class="train" src="https://js.cx/clipart/train.gif" onclick="this.style.left='450px'">

</body>

</html>
```

The CSS transition is based on that curve:

```
.train {
  left: 0;
  transition: left 5s cubic-bezier(0, 0, 1, 1);
  /* click on a train sets left to 450px, thus triggering the animation */
}
```

…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):
Resultstyle.cssindex.html.train {
  position: relative;
  cursor: pointer;
  width: 177px;
  height: 160px;
  left: 0px;
  transition: left 5s cubic-bezier(0.0, 0.5, 0.5, 1.0);
}<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  <img class="train" src="https://js.cx/clipart/train.gif" onclick="this.style.left='450px'">

</body>

</html>CSS:

    .train {
  left: 0;
  transition: left 5s cubic-bezier(0, .5, .5, 1);
  /* click on a train sets left to 450px, thus triggering the animation */
}

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, which we described above.
Other names are shorthands for the following cubic-bezier:

ease*
ease-in
ease-out
ease-in-out

(0.25, 0.1, 0.25, 1.0)
(0.42, 0, 1.0, 1.0)
(0, 0, 0.58, 1.0)
(0.42, 0, 0.58, 1.0)

* – 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;
  /* same as transition: left 5s cubic-bezier(0, .5, .5, 1); */
}
```

But it looks a bit differently.
A Bezier curve can make the animation exceed its range.
The control points on the curve can have any y coordinates: even negative or huge ones. Then the Bezier curve would also extend 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);
  /* click on a train sets left to 450px */
}
```

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.

Resultstyle.cssindex.html.train {
  position: relative;
  cursor: pointer;

```
  width: 177px;
  height: 160px;
  left: 100px;
  transition: left 5s cubic-bezier(.5, -1, .5, 2);
}<!doctype html>
<html>

<head>
  <meta charset="UTF-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  <img class="train" src="https://js.cx/clipart/train.gif" onclick="this.style.left='400px'">

</body>

</html>
```
Why it happens is 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 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 beyond the starting left and y>1 – past 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 do we make a Bezier curve for a specific task? There are many tools.

For instance, we can do it on the site https://cubic-bezier.com.
Browser developer tools also have special support for Bezier curves in CSS:

Open the developer tools with F12 (Mac: Cmd+Opt+I).
Select the Elements tab, then pay attention to the Styles sub-panel at the right side.
CSS properties with a word cubic-bezier will have an icon before this word.
Click this icon to edit the curve.

StepsThe timing function steps(number of steps[, start/end]) allows splitting an

transition into multiple steps.

Let's see that in an example with digits.

Here's a list of digits, without any animations, just as a source:

Resultstyle.cssindex.html#digit {

```
#digit {
  border: 1px solid red;
  width: 1.2em;
}

#stripe {
  display: inline-block;
  font: 32px monospace;
}
```
<!DOCTYPE html>
```html
<html>

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  <div id="digit"><div id="stripe">0123456789</div></div>

</body>
</html>
```
In the HTML, a stripe of digits is enclosed into a fixed-length <div id="digits">:

```html
    <div id="digit">
  <div id="stripe">0123456789</div>
</div>
```

The #digit div has a fixed width and a border, so it looks like a red window.

We'll make a timer: the digits will appear one by one, in a discrete way.

To achieve that, we'll hide the #stripe outside of #digit using overflow: hidden, and then shift the #stripe to the left step-by-step.

There will be 9 steps, a step-move for each digit:

```css
    #stripe.animate  {
  transform: translate(-90%);
```

```
  transition: transform 9s steps(9, start);
}
```

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 make the first step immediately.

In action:

Resultstyle.cssindex.html#digit {

```
#digit {
  width: .5em;
  overflow: hidden;
  font: 32px monospace;
  cursor: pointer;
}

#stripe {
  display: inline-block
}

#stripe.animate {
  transform: translate(-90%);
  transition-property: transform;
  transition-duration: 9s;
  transition-timing-function: steps(9, start);
}
```

```html
<!DOCTYPE html>
<html>

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  Click below to animate:

  <div id="digit"><div id="stripe">0123456789</div></div>

  <script>
    digit.onclick = function() {
      stripe.classList.add('animate');
    }
```

```
  </script>
```

```
</body>
```

```
</html>
```
A click on the digit changes it 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 – -90%
(the last second shows the final value).

Here, the first change was immediate because of start in the steps.
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 for steps(9, end) would go like this:

0s – 0 (during the first second nothing changes)
1s – -10% (first change at the end of the 1st second)
2s – -20%
…
9s – -90%

Here's steps(9, end) in action (note the pause before the first digit change):
Resultstyle.cssindex.html#digit {
  width: .5em;
  overflow: hidden;
  font: 32px monospace;
  cursor: pointer;
}

#stripe {
  display: inline-block
}

#stripe.animate {
  transform: translate(-90%);
  transition-property: transform;
  transition-duration: 9s;
  transition-timing-function: steps(9, end);
}<!DOCTYPE html>
<html>

```
<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
</head>

<body>

  Click below to animate:

  <div id="digit"><div id="stripe">0123456789</div></div>

  <script>
    digit.onclick = function() {
      stripe.classList.add('animate');
    }
  </script>


</body>

</html>
```
There are also some pre-defined shorthands for steps(...):

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, as they represent not a real animation, but rather a single-step change. We mention them here for completeness.
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 sail there and back when clicked, each time farther and farther to the right:

The animation is initiated by the function go that re-runs each time the transition finishes, and flips the direction:

```
      boat.onclick = function() {
  //...
  let times = 1;

  function go() {
    if (times % 2) {
      // sail to the right
      boat.classList.remove('back');
      boat.style.marginLeft = 100 * times + 200 + 'px';
    } else {
      // sail 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 a 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.

KeyframesWe 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 can attach the animation to the element and specify additional parameters for it.
Here's an example with explanations:

```html
    <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.
You probably won't need @keyframes often, unless everything is in constant motion on your sites.
PerformanceMost CSS properties can be animated, because most of them are numeric values. For instance, width, color, font-size are all numbers. When you animate them, the browser gradually changes these numbers frame by frame, creating a smooth effect. However, not all animations will look as smooth as you'd like, because different CSS properties cost differently to change.

In more technical details, when there's a style change, the browser goes through 3 steps to render the new look:

Layout: re-compute the geometry and position of each element, then
Paint: re-compute how everything should look like at their places, including background, colors,
Composite: render the final results into pixels on screen, apply CSS transforms if they exist.

During a CSS animation, this process repeats every frame. However, CSS properties that never affect geometry or position, such as color, may skip the Layout step. If a color changes, the browser doesn't calculate any new geometry, it goes to Paint !' Composite. And there are few properties that directly go to Composite. You can find a longer list of CSS properties and which stages they trigger at https://csstriggers.com.
The calculations may take time, especially on pages with many elements and a complex layout. And the delays are actually visible on most devices, leading to "jittery", less fluid animations.
Animations of properties that skip the Layout step are faster. It's even better if Paint is skipped too.
The transform property is a great choice, because:

CSS transforms affect the target element box as a whole (rotate, flip, stretch, shift it).
CSS transforms never affect neighbour elements.

…So browsers apply transform "on top" of existing Layout and Paint calculations, in the Composite stage.
In other words, the browser calculates the Layout (sizes, positions), paints it with colors, backgrounds, etc at the Paint stage, and then applies transform to element boxes that need it.
Changes (animations) of the transform property never trigger Layout and Paint steps.
More than that, the browser leverages the graphics accelerator (a special chip on the CPU or graphics card) for CSS transforms, thus making them very efficient.
Luckily, the transform property is very powerful. By using transform on an element, you could rotate and flip it, stretch and shrink it, move it around, and much more. So instead of left/margin-left properties we can use transform: translateX(…), use transform: scale for increasing element size, etc.
The opacity property also never triggers Layout (also skips Paint in Mozilla Gecko). We can use it for show/hide or fade-in/fade-out effects.
Paring transform with opacity can usually solve most of our needs, providing fluid, good-looking animations.
For example, here clicking on the #boat element adds the class with transform: translateX(300) and opacity: 0, thus making it move 300px to the right and disappear:

```html
<img src="https://js.cx/clipart/boat.png" id="boat">

<style>
#boat {
  cursor: pointer;
  transition: transform 2s ease-in-out, opacity 2s ease-in-out;
}

.move {
  transform: translateX(300px);
  opacity: 0;
}
</style>
<script>
  boat.onclick = () => boat.classList.add('move');
</script>
```

Here's a more complex example, with @keyframes:

```html
<h2 onclick="this.classList.toggle('animated')">click me to start / stop</h2>
<style>
  .animated {
    animation: hello-goodbye 1.8s infinite;
```

```
    width: fit-content;
  }
  @keyframes hello-goodbye {
   0% {
     transform: translateY(-60px) rotateX(0.7turn);
     opacity: 0;
   }
   50% {
     transform: none;
     opacity: 1;
   }
   100% {
     transform: translateX(230px) rotateZ(90deg) scale(0.5);
     opacity: 0;
   }
  }
</style>
```

SummaryCSS animations allow smoothly (or step-by-step) animated 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:

MeritsSimple things done simply.

Fast and lightweight for CPU.

DemeritsJavaScript 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 as part of the animation.

In early examples in this chapter, we animate font-size, left, width, height, etc. In real life projects, we should use transform: scale() and transform: translate() for better performance.

The majority of animations can be implemented using CSS as described in this chapter. And the transitionend event allows JavaScript to be run 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.

TasksAnimate a plane (CSS)importance: 5Show 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.solutionCSS 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.Animate the flying plane (CSS)importance: 5Modify the solution of the previous task Animate a plane (CSS) to make the plane grow more than its 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.
solutionWe 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.Animated circleimportance: 5Create 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:

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.solutionOpen the solution in a sandbox.Animated circle with callbackIn the task Animated circle an animated growing circle is shown.
Now let's say we need not just a circle, but to show a message inside it. The message should appear after the animation is complete (the circle is fully grown), otherwise it would look ugly.
In the solution of the task, the function showCircle(cx, cy, radius) draws the circle, but gives no way to track when it's ready.
Add a callback argument: showCircle(cx, cy, radius, callback) to be called when the animation is complete. The callback should receive the circle <div> as an argument. Here's the example:

```
showCircle(150, 150, 100, div => {
  div.classList.add('message-ball');
  div.append("Hello, world!");
});
```

Demo:

Take the solution of the task Animated circle as the base.
solutionOpen the solution in a sandbox.Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To

insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/css-animations","identifier":"\/css-animations"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialAnimation{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Animation","item":"https://javascript.info/animation"}]}October 14, 2022JavaScript animationsJavaScript 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 setIntervalAn 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:
Resultindex.html
```
<!DOCTYPE HTML>
<html>

<head>
  <style>
    #train {
      position: relative;
      cursor: pointer;
    }
  </style>
</head>

<body>

  <img id="train" src="https://js.cx/clipart/train.gif">


  <script>
    train.onclick = function() {
      let start = Date.now();

      let timer = setInterval(function() {
        let timePassed = Date.now() - start;

        train.style.left = timePassed / 5 + 'px';

        if (timePassed > 2000) clearInterval(timer);

      }, 20);
    }
  </script>
```

```
</body>
```

```
</html>
```
Using requestAnimationFrameLet'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 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 milliseconds. 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 animationNow 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;
}
```

Its graph:

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:
Resultanimate.jsindex.htmlfunction animate({duration, draw, timing}) {

```
  let start = performance.now();

  requestAnimationFrame(function animate(time) {
    let timeFraction = (time - start) / duration;
    if (timeFraction > 1) timeFraction = 1;
```

```
    let progress = timing(timeFraction)

    draw(progress);

    if (timeFraction < 1) {
      requestAnimationFrame(animate);
    }

  });
}
```
```html
<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
  <style>
    progress {
      width: 5%;
    }
  </style>
  <script src="animate.js"></script>
</head>

<body>


  <progress id="elem"></progress>

  <script>
    elem.onclick = function() {
      animate({
        duration: 1000,
        timing: function(timeFraction) {
          return timeFraction;
        },
        draw: function(progress) {
          elem.style.width = progress * 100 + '%';
        }
      });
    };
  </script>


</body>

</html>
```
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 functionsWe 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 nIf 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 arcFunction:

```
function circ(timeFraction) {
  return 1 - Math.sin(Math.acos(timeFraction));
}
```

The graph:

Back: bow shootingThis 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:

BounceImagine 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; 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 animationOne 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.
easeOutIn 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:
Resultstyle.cssindex.html#brick {
  width: 40px;
  height: 20px;
  background: #EE6B47;
  position: relative;
  cursor: pointer;
}

#path {
  outline: 1px solid #E8C48E;
  width: 540px;
  height: 20px;
}<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
  <script src="https://js.cx/libs/animate.js"></script>
</head>

<body>

  <div id="path">
    <div id="brick"></div>
  </div>

  <script>
    function makeEaseOut(timing) {
      return function(timeFraction) {
        return 1 - timing(1 - timeFraction);
```

```
      }
    }

    function bounce(timeFraction) {
      for (let a = 0, b = 1; 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)
        }
      }
    }

    let bounceEaseOut = makeEaseOut(bounce);

    brick.onclick = function() {
      animate({
        duration: 3000,
        timing: bounceEaseOut,
        draw: function(progress) {
          brick.style.left = progress * 500 + 'px';
        }
      });
    };
  </script>


</body>

</html>
```

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.

easeInOutWe 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:
```
Resultstyle.cssindex.html#brick {
  width: 40px;
  height: 20px;
  background: #EE6B47;
  position: relative;
  cursor: pointer;
}

#path {
  outline: 1px solid #E8C48E;
  width: 540px;
  height: 20px;
}
```
```
<!DOCTYPE HTML>
<html>

<head>
```

```
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
  <script src="https://js.cx/libs/animate.js"></script>
</head>

<body>


  <div id="path">
    <div id="brick"></div>
  </div>

  <script>
    function makeEaseInOut(timing) {
      return function(timeFraction) {
        if (timeFraction < .5)
          return timing(2 * timeFraction) / 2;
        else
          return (2 - timing(2 * (1 - timeFraction))) / 2;
      }
    }


    function bounce(timeFraction) {
      for (let a = 0, b = 1; 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)
        }
      }
    }

    let bounceEaseInOut = makeEaseInOut(bounce);

    brick.onclick = function() {
      animate({
        duration: 3000,
        timing: bounceEaseInOut,
        draw: function(progress) {
          brick.style.left = progress * 500 + 'px';
        }
      });
    };
  </script>


</body>
```

</html>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:
Resultstyle.cssindex.htmltextarea {
  display: block;
  border: 1px solid #BBB;
  color: #444;
  font-size: 110%;
}

button {
  margin-top: 10px;
}<!DOCTYPE HTML>
<html>

<head>
  <meta charset="utf-8">
  <link rel="stylesheet" href="style.css">
  <script src="https://js.cx/libs/animate.js"></script>
</head>

<body>

  <textarea id="textExample" rows="5" cols="60">He took his vorpal sword in hand:
Long time the manxome foe he sought—
So rested he by the Tumtum tree,
And stood awhile in thought.

```
    </textarea>

    <button onclick="animateText(textExample)">Run the animated typing!</button>

    <script>
      function animateText(textArea) {
        let text = textArea.value;
        let to = text.length,
          from = 0;

        animate({
          duration: 5000,
          timing: bounce,
          draw: function(progress) {
            let result = (to - from) * progress + from;
            textArea.value = text.slice(0, Math.ceil(result))
          }
        });
      }


      function bounce(timeFraction) {
        for (let a = 0, b = 1; 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)
          }
        }
      }
    </script>


  </body>
```

```
</html>
```
SummaryFor 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.
TasksAnimate the bouncing ballimportance: 5Make a bouncing ball. Click to see how it
should look:

  Open a sandbox for the task.solutionTo 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 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.Animate the ball bouncing to the rightimportance: 5Make 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.
solutionIn 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.Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/js-animation","identifier":"\/js-animation"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

Tutorial{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}]}Web componentsWeb 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 heightCustom elementsShadow DOMTemplate elementShadow DOM slots, compositionShadow DOM stylingShadow DOM and eventsCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial map

TutorialWeb components{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Web components","item":"https://javascript.info/ web-components"}]}June 26, 2022From the orbital heightThis 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 are such complex devices created?
Which principles could we borrow to make our development same-level reliable and scalable? Or, at least, close to it?
Component architectureThe 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:

Top navigation.
User info.
Follow suggestions.
Submit form.
(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 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.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:Vjavascript.infoVwebcomponents-intro","identifier":"Vwebcomponents-intro"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialWeb components{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Web components","item":"https://javascript.info/
web-components"}]}March 26, 2021Custom elementsWe can create custom HTML elements, described by our class, with its own methods and properties, events and so on.
Once a 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:

Autonomous custom elements – "all-new" elements, extending the abstract
HTMLElement class.
Customized built-in elements – extending built-in elements, like a customized button,
based on HTMLButtonElement etc.

First we'll cover autonomous elements, and then move to 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 this method when the element is added to the document
  // (can be called many times if an element is repeatedly added/removed)
}

disconnectedCallback() {
  // browser calls this method 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.

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>
```

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.
We need to register our new element by customElements.define(tag, class).
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.

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 is created, but 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 attributesIn 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 other, unlisted attributes (that's 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-zone-name'];
  }

  attributeChangedCallback(name, oldValue, newValue) { // (4)
    this.render();
  }

}

customElements.define("time-formatted", TimeFormatted);
</script>

<time-formatted id="elem" hour="numeric" minute="numeric" second="numeric"></time-formatted>

<script>
setInterval(() => elem.setAttribute('datetime', new Date()), 1000); // (5)
</script>
```

The rendering logic is moved to render() helper method.

We call it once when the element is inserted into page.
For a change of an attribute, listed in observedAttributes(), attributeChangedCallback triggers.
…and re-renders the element.
At the end, we can easily make a live timer.

Rendering orderWhen 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 is going to 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:

outer connected.
inner connected.
outer initialized.
inner initialized.

We can clearly see that the outer element finishes initialization (3) before the inner one (4).
There's no built-in callback that triggers after nested elements are ready. If needed, 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 elementsNew 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 HTML elements by inheriting from their classes. For example, buttons are instances of HTMLButtonElement, let's build upon it.

Extend HTMLButtonElement with our class:

```
class HelloButton extends HTMLButtonElement { /* custom element methods */ }
```

Provide the third argument to customElements.define, that specifies the tag:

```
customElements.define('hello-button', HelloButton, {extends: 'button'});
```

There may be different tags that share the same DOM-class, that's why specifying extends is needed.

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 HTMLButtonElement {
  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.
Compatiblity: https://caniuse.com/#feat=custom-elementsv1.

SummaryCustom elements can be of two types:

"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> */
```

"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. There's a polyfill https://
github.com/webcomponents/polyfills/tree/master/packages/webcomponentsjs.
TasksLive timer elementWe already have <time-formatted> element to show a nicely
formatted time.
Create <live-timer> element to show the current time:

It should use <time-formatted> internally, not duplicate its functionality.
Ticks (updates) every second.
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:

Open a sandbox for the task.solutionPlease note:

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.
We can access current date as elem.date property. All class methods and properties

are naturally element methods and properties.

TutorialWeb components{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Web components","item":"https://javascript.info/web-components"}]}December 6, 2020Shadow DOMShadow 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 DOMDid 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:

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 and other related specifications.

Shadow treeA DOM element can have two types of DOM subtrees:

Light tree – a regular DOM subtree, made of HTML children. All subtrees that we've seen in previous chapters were "light".

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>
```

That's how the resulting DOM looks in Chrome dev tools, all the content is under "#shadow-root":

First, the call to elem.attachShadow({mode: …}) creates a shadow tree. There are two limitations:

We can create only one shadow root per element.
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 <img>, 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
```

EncapsulationShadow DOM is strongly delimited from the main document:

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.
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>
```

The style from the document does not affect the shadow tree.
…But the style from the inside works.
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.

SummaryShadow DOM is a way to create a component-local DOM.

shadowRoot = elem.attachShadow({mode: open|closed}) – creates shadow DOM for
elem. If mode="open", then it's accessible as elem.shadowRoot property.
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.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before
commenting…If you have suggestions what to improve - please submit a GitHub issue
or a pull request instead of commenting.If you can't understand something in the article
– please elaborate.To insert few words of code, use the <code> tag, for several lines –
wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin,
codepen…)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https:\/\/javascript.info\/shadow-dom","identifier":"\/
shadow-dom"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialWeb components{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Web components","item":"https://javascript.info/
web-components"}]}April 14, 2022Template elementA 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 templateThe 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:

```html
<template id="tmpl">
  <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(tmpl.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>:

```html
<template id="tmpl">
```

```
  <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(tmpl.content.cloneNode(true)); // (*)

    elem.shadowRoot.getElementById('message').innerHTML = "Hello from the
shadows!";
  };
</script>
```

In the line (*) when we clone and insert tmpl.content, as its 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>
```

SummaryTo 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 use of 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.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/template-element","identifier":"\/template-element"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialWeb components{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Web components","item":"https://javascript.info/web-components"}]}September 27, 2022Shadow DOM slots, compositionMany 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 slotsLet'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>
```

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 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>
```

We created the shadow DOM, so here it is, under #shadow-root. 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:

The result is called "flattened" DOM:

```
<user-card>
#shadow-root
  <div>Name:
    <slot name="username">
      <!-- slotted element is inserted into the slot -->
      <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 exists only for rendering and event-handling purposes. It's kind of "virtual". That's how things are shown. But the nodes in the document are actually not moved around!
That can be easily checked if we run querySelectorAll: nodes are still at their places.

```
    // light DOM <span> nodes are still at the same place, under `<user-card>`
alert( document.querySelectorAll('user-card span').length ); // 2
```

So, the flattened DOM is derived from shadow DOM by inserting slots. The browser renders it and uses for style inheritance, event propagation (more about that later). 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 <span> here is ignored (as it's not a top-level child of <user-card>):

```
    <user-card>
  <span slot="username">John Smith</span>
  <div>
    <!-- invalid slot, must be direct child of user-card -->
    <span slot="birthday">01.01.2001</span>
  </div>
</user-card>
```

If there are multiple elements in light DOM with the same slot name, they are appended into the slot, one after another.
For example, this:

```
    <user-card>
  <span slot="username">John</span>
  <span slot="username">Smith</span>
</user-card>
```

Gives this flattened DOM with two elements in <slot name="username">:

```
    <user-card>
 #shadow-root
  <div>Name:
   <slot name="username">
     <span slot="username">John</span>
     <span slot="username">Smith</span>
   </slot>
  </div>
  <div>Birthday:
   <slot name="birthday"></slot>
  </div>
</user-card>
```

Slot fallback contentIf we put something inside a <slot>, it becomes the fallback, "default" 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: first unnamedThe 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 shows all 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>
```

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>Other information</legend>
    <slot>
     <div>I like to swim.</div>
     <div>...And play volleyball too!</div>
    </slot>
   </fieldset>
</user-card>
```

Menu exampleNow 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="tmpl">
  <style> /* menu styles */ </style>
  <div class="menu">
    <slot name="title"></slot>
    <ul><slot name="item"></slot></ul>
  </div>
</template>
```

<span slot="title"> goes into <slot name="title">.
There are many <li slot="item"> in the <custom-menu>, but only one <slot name="item"> in the template. So all such <li slot="item"> are 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, <li> must be a direct child of <ul>. 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:

Of course, we can add more functionality to it: events, methods and so on.

Updating slotsWhat 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 code 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:

At initialization:

slotchange: title triggers immediately, as the slot="title" from the light DOM gets into the corresponding slot.

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 APIFinally, 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>`;

    // triggers when slot content changes
    this.shadowRoot.firstElementChild.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>
```

SummaryUsually, if an element has shadow DOM, then its light DOM is not displayed. Slots allow to show elements from light DOM in specified places of 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 property, 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 know how to show elements from light DOM in 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.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/slots-composition","identifier":"\/slots-composition"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialWeb components{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Web components","item":"https://javascript.info/web-components"}]}May 17, 2021Shadow DOM stylingShadow DOM may include both <style> and <link rel="stylesheet" href="…"> tags. In the latter case, stylesheets are HTTP-cached, so they are not redownloaded for multiple components that use same template.
As a general rule, local styles work only inside the shadow tree, and document styles work outside of it. But there are few exceptions.
:hostThe :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:

```html
    <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>
```

CascadingThe shadow host (<custom-dialog> itself) resides in the light DOM, so it's affected by document CSS rules.
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" component styles in its :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="tmpl">
 <style>
  :host([centered]) {
    position: fixed;
    left: 50%;
    top: 50%;
    transform: translate(-50%, -50%);
    border-color: blue;
  }

  :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>
```

Now the additional centering styles are only applied to the first dialog: <custom-dialog centered>.

To summarize, we can use :host-family of selectors to style the main element of the component. These styles (unless !important) can be overridden by the document.

Styling slotted contentNow 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 <span> 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>
```

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>
```

Here <p>John Smith</p> becomes bold, because CSS inheritance is in effect between the <slot> and its contents. But in CSS itself not all properties are inherited. Another option is to use ::slotted(selector) pseudo-class. It matches elements based on two conditions:

That's a slotted element, that comes from the light DOM. Slot name doesn't matter. Just any slotted element, but only the element itself, not its children.
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>
```

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 propertiesHow do we style internal elements of a component from the main document?
Selectors like :host apply rules to <custom-dialog> element or <user-card>, but how to style shadow DOM elements inside them?
There's no selector that can directly affect shadow DOM styles from the document. But 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, and the outer document can set its value:

```
  <style>
  .field {
    color: var(--user-card-field-color, black);
    /* if --user-card-field-color is not defined, use black color */
  }
</style>
<div class="field">Name: <slot name="username"></slot></div>
<div class="field">Birthday: <slot name="birthday"></slot></div>
```

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="tmpl">
  <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('tmpl').content.cloneNode(true));
  }
});
</script>

<user-card>
  <span slot="username">John Smith</span>
  <span slot="birthday">01.01.2001</span>
</user-card>
```

SummaryShadow DOM can include styles, such as <style> or <link rel="stylesheet">.
Local styles can affect:

shadow tree,
shadow host with :host and :host() 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 lives in the outer document)
slotted elements and their contents (as that's also 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:

The component uses a custom CSS property to style key elements, such as var(--component-name-title, <default value>).

Component author publishes these properties for developers, they are same important as other public component methods.
When a developer wants to style a title, they assign --component-name-title CSS property for the shadow host or above.
Profit!

Sharevar disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/shadow-dom-style","identifier":"\/shadow-dom-style"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialWeb components{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Web components","item":"https://javascript.info/web-components"}]}July 15, 2019Shadow DOM and eventsThe 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>
```

If you click on the button, the messages are:

Inner target: BUTTON – internal event handler gets the correct target, the element inside shadow DOM.
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 to 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 <span slot="username"> 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>
```

If a click happens on "John Smith", for both inner and outer handlers the target is `<span slot="username">`. 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 `<b>Name</b>`, 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>
```

So, for a click on <span slot="username">, 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.composedMost 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 eventsWhen 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>
```

SummaryEvents 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.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https:\/\/javascript.info\/shadow-dom-events","identifier":"\/shadow-dom-events"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
Tutorial{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}]}
Regular expressionsRegular expressions is a powerful way of doing search and replace in strings.
Patterns and flagsCharacter classesUnicode: flag "u" and class \p{...}Anchors: string start ^ and end $Multiline mode of anchors ^ $, flag "m"Word boundary: \bEscaping, special charactersSets and ranges [...]Quantifiers +, *, ? and {n}Greedy and lazy quantifiersCapturing groupsBackreferences in pattern: \N and \k<name>Alternation (OR) |Lookahead and lookbehindCatastrophic backtrackingSticky flag "y", searching at positionMethods of RegExp and StringCtrl + !•Previous lessonCtrl + !'Next lessonShareTutorial map
TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}December 7, 2020Patterns and flagsRegular expressions are patterns that provide a powerful way to search and replace in text.
In JavaScript, they are available via the RegExp object, as well as being integrated in methods of strings.
Regular ExpressionsA regular expression (also "regexp", or just "reg") consists of a pattern and optional flags.
There are two syntaxes that can be used to create a regular expression object.
The "long" syntax:

```
regexp = 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.

In both cases regexp becomes an instance of the built-in RegExp class.

The main difference between these two syntaxes is that pattern using slashes /.../ does not allow for expressions to be inserted (like string template literals with ${...}). They are fully static.

Slashes are used when we know the regular expression at the code writing time – and that's the most common situation. While new RegExp is more often used when we need to create a regexp "on the fly" from a dynamically generated string. For instance:

```
let tag = prompt("What tag do you want to find?", "h2");

let regexp = new RegExp(`<${tag}>`); // same as /<h2>/ if answered "h2" in the prompt above
```

FlagsRegular 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 match is returned.
m

Multiline mode (covered in the chapter Multiline mode of anchors ^ $, flag "m").
s
Enables "dotall" mode, that allows a dot . to match newline character \n (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" and class \p{...}.
y
"Sticky" mode: searching at the exact position in the text (covered in the chapter Sticky flag "y", searching at position)

Colors
From here on the color scheme is:

regexp – red
string (where we search) – blue
result – green

Searching: str.matchAs mentioned previously, regular expressions are integrated with string methods.
The method str.match(regexp) finds all matches of regexp in the string str.
It has 3 working modes:

If the regular expression has flag g, it returns an array of all matches:

```
let str = "We will, we will rock you";

alert( str.match(/we/gi) ); // We,we (an array of 2 substrings that match)
```

Please note that both We and we are found, because flag i makes the regular

expression case-insensitive.

If there's no such flag it returns only the first match in the form of an array, with the full match at index 0 and some additional details in properties:

```
let str = "We will, we will rock you";

let result = str.match(/we/i); // without flag g

alert( result[0] );     // We (1st match)
alert( result.length ); // 1

// Details:
alert( result.index );  // 0 (position of the match)
alert( result.input );  // We will, we will rock you (source string)
```

The array may have other indexes, besides 0 if a part of the regular expression is enclosed in parentheses. We'll cover that in the chapter Capturing groups.

And, finally, if there are no matches, null is returned (doesn't matter if there's flag g or not).

This a very important nuance. If there are no matches, we don't receive an empty array, but instead receive null. Forgetting about that may lead to errors, e.g.:

```
let matches = "JavaScript".match(/HTML/); // = null
```

```
if (!matches.length) { // Error: Cannot read property 'length' of null
  alert("Error in the line above");
}
```

If we'd like the result to always be an array, we can write it this way:

```
let matches = "JavaScript".match(/HTML/) || [];
```

```
if (!matches.length) {
  alert("No matches"); // now it works
}
```

Replacing: str.replaceThe method str.replace(regexp, replacement) replaces matches found using regexp in string str with replacement (all matches if there's flag g, otherwise, only the first one).
For instance:

```
        // no flag g
alert( "We will, we will".replace(/we/i, "I") ); // I will, we will

// with flag g
alert( "We will, we will".replace(/we/ig, "I") ); // I will, I will
```

The second argument is the replacement string. We can use special character combinations in it to insert fragments of the match:

Symbols
Action in the replacement string

$&
inserts the whole match

$`
inserts a part of the string before the match

$'
inserts a part of the string after the match

$n
if n is a 1-2 digit number, then it inserts the contents of n-th parentheses, more about it in the chapter Capturing groups

$<name>
inserts the contents of the parentheses with the given name, more about it in the chapter Capturing groups

`$$`

inserts character $

An example with $&:

```
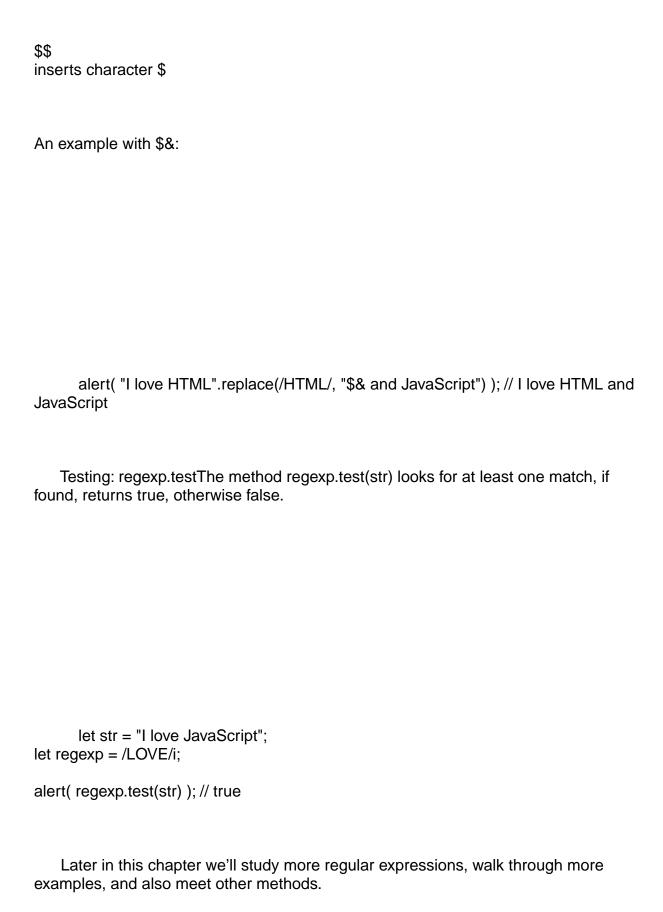alert( "I love HTML".replace(/HTML/, "$& and JavaScript") ); // I love HTML and JavaScript
```

Testing: regexp.testThe method regexp.test(str) looks for at least one match, if found, returns true, otherwise false.

```
let str = "I love JavaScript";
let regexp = /LOVE/i;

alert( regexp.test(str) ); // true
```

Later in this chapter we'll study more regular expressions, walk through more examples, and also meet other methods.

Full information about the methods is given in the article Methods of RegExp and String.

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.match(regexp) looks for matches: all of them if there's g flag, otherwise, only the first one.

The method str.replace(regexp, replacement) replaces matches found using regexp with replacement: all of them if there's g flag, otherwise only the first one.

The method regexp.test(str) returns true if there's at least one match, otherwise, it returns false.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-introduction","identifier":"\/regexp-introduction"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}December 12, 2021Character classesConsider a practical task – we have a phone number like "+7(903)-123-45-67", and we need to turn it into pure numbers: 79031234567.

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 the "digit" class. It's written as \d and corresponds to "any single digit".

For instance, let's find the first digit in the phone number:

```
    let str = "+7(903)-123-45-67";

let regexp = /\d/;

alert( str.match(regexp) ); // 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 regexp = /\d/g;

alert( str.match(regexp) ); // array of matches: 7,9,0,3,1,2,3,4,5,6,7

// let's make the digits-only phone number of them:
alert( str.match(regexp).join('') ); // 79031234567
```

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: includes spaces, tabs \t, newlines \n and few other rare characters, such as \v, \f and \r.
\w ("w" is from "word")
A "wordly" character: either a letter of Latin 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", such as 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 = "Is there CSS4?";
let regexp = /CSS\d/

alert( str.match(regexp) ); // CSS4
```

Also we can use many character classes:

```
        alert( "I love HTML5!".match(/\s\w\w\w\w\d/) ); // ' HTML5'
```

The match (each regexp character class has the corresponding result character):

Inverse classesFor every character class there exists an "inverse class", denoted with the same letter, but uppercased.

The "inverse" 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, e.g a non-latin letter or a space.

In the beginning of the chapter we saw how to make a number-only phone number from a string like +7(903)-123-45-67: find 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
```

A dot is "any character"A dot . is a special character class that matches "any character except a newline".
For instance:

```
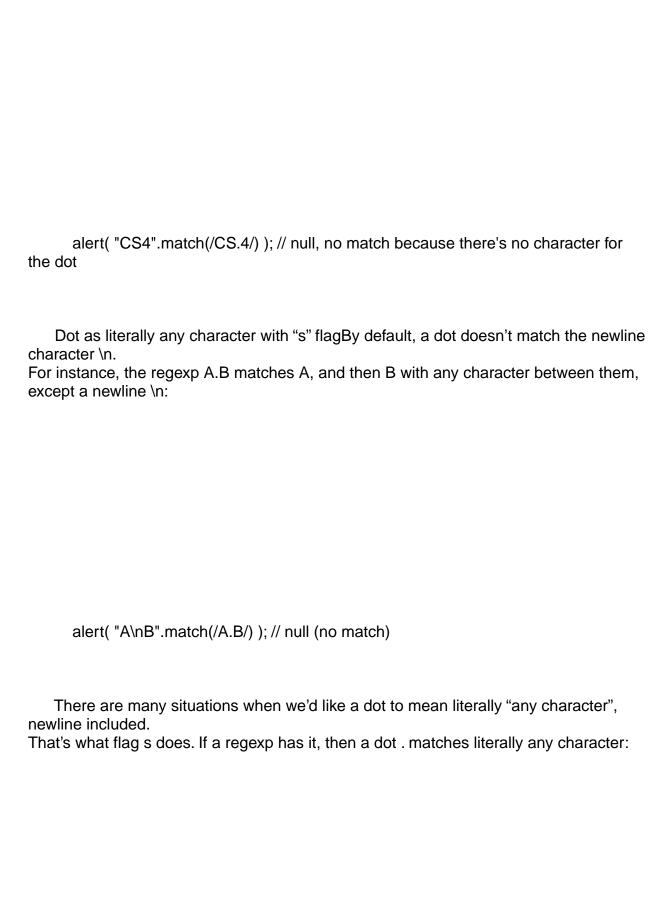alert( "Z".match(/./) ); // Z
```

Or in the middle of a regexp:

```
let regexp = /CS.4/;

alert( "CSS4".match(regexp) ); // CSS4
alert( "CS-4".match(regexp) ); // CS-4
alert( "CS 4".match(regexp) ); // CS 4 (space is also a character)
```

Please note that a dot means "any character", but not the "absence 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

Dot as literally any character with "s" flagBy default, a dot doesn't match the newline character \n.
For instance, the regexp A.B matches A, and then B with any character between them, except a newline \n:

```
alert( "A\nB".match(/A.B/) ); // null (no match)
```

There are many situations when we'd like a dot to mean literally "any character", newline included.
That's what flag s does. If a regexp has it, then a dot . matches literally any character:

```
alert( "A\nB".match(/A.B/s) ); // A\nB (match!)
```

Not supported in IE
The s flag is not supported in IE.
Luckily, there's an alternative, that works everywhere. We can use a regexp like [\s\S] to match "any character" (this pattern will be covered in the article Sets and ranges [...]).

```
alert( "A\nB".match(/A[\s\S]B/) ); // A\nB (match!)
```

The pattern [\s\S] literally says: "a space character OR not a space character". In other words, "anything". We could use another pair of complementary classes, such as [\d\D], that doesn't matter. Or even the [^] – as it means match any character except nothing.
Also we can use this trick if we want both kind of "dots" in the same pattern: the actual dot . behaving the regular way ("not including a newline"), and also a way to match "any character" with [\s\S] or alike.

Pay attention to spaces
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 hyphen:

```
alert( "1 - 5".match(/\d-\d/) ); // null, no match!
```

Let's fix it adding spaces into the regexp \d - \d:

```
alert( "1 - 5".match(/\d - \d/) ); // 1 - 5, now it works
// or we can use \s class:
alert( "1 - 5".match(/\d\s-\s\d/) ); // 1 - 5, also works
```

A space is a character. Equal in importance with any other character.
We can't add or remove spaces from a regular expression and expect it to work the same.
In other words, in a regular expression all characters matter, spaces too.

SummaryThere exist following character classes:

\d – digits.
\D – non-digits.
\s – space symbols, tabs, newlines.
\S – all but \s.
\w – Latin letters, digits, underscore '_'.

\W – all but \w.

. – any character if with the regexp 's' flag, otherwise any except a newline \n.

…But that's not all!

Unicode encoding, used by JavaScript for strings, provides many properties for characters, like: which language the letter belongs to (if it's a letter), is it a punctuation sign, etc.

We can search by these properties as well. That requires flag u, covered in the next article.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-character-classes","identifier":"\/regexp-character-classes"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}August 5, 2022Unicode: flag "u" and class \p{...} JavaScript uses Unicode encoding for strings. Most characters are encoded with 2 bytes, but that allows to represent at most 65536 characters.

That range is not big enough to encode all possible characters, that's why some rare characters are encoded with 4 bytes, for instance like Ø5Ü³ (mathematical X) or Ø=Þ (a smile), some hieroglyphs and so on.

Here are the Unicode values of some characters:

Character
Unicode
Bytes count in Unicode

a
0x0061
2

"H
0x2248

2

Ø5Ü³
0x1d4b3
4

Ø5Ü´
0x1d4b4
4

Ø=Þ
0x1f604
4

So characters like a and "H occupy 2 bytes, while codes for Ø5Ü³, Ø5Ü´ and Ø=Þ are longer, they hav
bytes.
Long time ago, when JavaScript language was created, Unicode encoding was simpler:
there were no 4-byte characters. So, some language features still handle them
incorrectly.
For instance, length thinks that here are two characters:

```
alert('Ø=Þ '.length); // 2
alert('Ø5Ü³'.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", you can read about them in the article Strings).
By default, regular expressions also treat 4-byte "long characters" as a pair of 2-byte

ones. And, as it happens with strings, that may lead to odd results. We'll see that a bit later, in the article Sets and ranges [...].

Unlike strings, regular expressions have flag u that fixes such problems. With such flag, a regexp handles 4-byte characters correctly. And also Unicode property search becomes available, we'll get to it next.

Unicode properties \p{…}Every character in Unicode has a lot of properties. They describe what "category" the character belongs to, contain miscellaneous information about it.

For instance, if a character has Letter property, it means that the character belongs to an alphabet (of any language). And Number property means that it's a digit: maybe Arabic or Chinese, and so on.

We can search for characters with a property, written as \p{…}. To use \p{…}, a regular expression must have flag u.

For instance, \p{Letter} denotes a letter in any language. We can also use \p{L}, as L is an alias of Letter. There are shorter aliases for almost every property.

In the example below three kinds of letters will be found: English, Georgian and Korean.

```
let str = "A  Ñ 11";

alert( str.match(/\p{L}/gu) ); // A, Ñ,11
alert( str.match(/\p{L}/g) ); // null (no matches, \p doesn't work without the flag "u")
```

Here's the main character categories and their subcategories:

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.

So, e.g. if we need letters in lower case, we can write \p{Ll}, punctuation signs: \p{P} and so on.
There are also other derived categories, like:

Alphabetic (Alpha), includes Letters L, plus letter numbers Nl (e.g. !k – a character for the roman number 12), plus some other symbols Other_Alphabetic (OAlpha).
Hex_Digit includes hexadecimal digits: 0-9, a-f.
…And so on.

Unicode supports many different properties, their full list would require a lot of space, so here are the references:

List all properties by a character: https://unicode.org/cldr/utility/character.jsp.
List all characters by a property: https://unicode.org/cldr/utility/list-unicodeset.jsp.
Short aliases for properties: https://www.unicode.org/Public/UCD/latest/ucd/PropertyValueAliases.txt.
A full base of Unicode characters in text format, with all properties, is here: https://www.unicode.org/Public/UCD/latest/ucd/.

Example: hexadecimal numbersFor instance, let's look for hexadecimal numbers, written as xFF, where F is a hex digit (0…9 or A…F).
A hex digit can be denoted as \p{Hex_Digit}:

```
let regexp = /x\p{Hex_Digit}\p{Hex_Digit}/u;

alert("number: xAF".match(regexp)); // xAF
```

Example: Chinese hieroglyphsLet's look for Chinese hieroglyphs.
There's a Unicode property Script (a writing system), that may have a value: Cyrillic,

Greek, Arabic, Han (Chinese) and so on, here's the full list.
To look for characters in a given writing system we should use Script=<value>, e.g. for Cyrillic letters: \p{sc=Cyrillic}, for Chinese hieroglyphs: \p{sc=Han}, and so on:

```
let regexp = /\p{sc=Han}/gu; // returns Chinese hieroglyphs

let str = `Hello Aô@C„2CTB O`Y} 123_456`;

alert( str.match(regexp) ); // O`,Y}
```

Example: currencyCharacters that denote a currency, such as $, €, ¥, have Unicode property \p{Currency_Symbol}, the short alias: \p{Sc}.
Let's use it to look for prices in the format "currency, followed by a digit":

```
let regexp = /\p{Sc}\d/gu;

let str = `Prices: $2, €1, ¥9`;

alert( str.match(regexp) ); // $2,€1,¥9
```

Later, in the article Quantifiers +, *, ? and {n} we'll see how to look for numbers that contain many digits.

SummaryFlag u enables the support of Unicode in regular expressions.

That means two things:

Characters of 4 bytes are handled correctly: as a single character, not two 2-byte characters.

Unicode properties can be used in the search: \p{…}.

With Unicode properties we can look for words in given languages, special characters (quotes, currencies) and so on.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-unicode","identifier":"\/regexp-unicode"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}April 13, 2022Anchors: string start ^ and end $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 $ – at the end.

For instance, let's test if the text starts with Mary:

```
let str1 = "Mary had a little lamb";
alert( /^Mary/.test(str1) ); // true
```

The pattern ^Mary means: "string start and then Mary".

Similar to this, we can test if the string ends with snow using snow$:

```
let str1 = "its fleece was white as snow";
alert( /snow$/.test(str1) ); // true
```

In these particular cases we could use string methods startsWith/endsWith instead. Regular expressions should be used for more complex tests.

Testing for a full matchBoth anchors together ^...$ are often used to test whether or not a string fully matches the pattern. For instance, to check if the user input is in the right format.

Let's check whether or not a string is a time in 12:34 format. That is: two digits, then a colon, and then another two digits.

In regular expressions language that's \d\d:\d\d:

```
let goodInput = "12:34";
let badInput = "12:345";

let regexp = /^\d\d:\d\d$/;
alert( regexp.test(goodInput) ); // true
alert( regexp.test(badInput) ); // false
```

Here the match for \d\d:\d\d must start exactly after the beginning of the text ^, and the end $ must immediately follow.

The whole string must be exactly in this format. If there's any deviation or an extra character, the result is false.

Anchors behave differently if flag m is present. We'll see that in the next article.

Anchors have "zero width"

Anchors ^ and $ 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).

TasksRegexp ^$Which string matches the pattern ^$?

solutionAn 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.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-anchors","identifier":"\/regexp-anchors"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true; TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}November 1, 2021Multiline mode of anchors ^ $, 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 the end of the string, but also at start/end of line.

Searching at line start ^In the example below the text has multiple lines. The pattern /^\d/gm takes a digit from the beginning of each line:

```
        let str = `1st place: Winnie
2nd place: Piglet
3rd place: Eeyore`;

console.log( str.match(/^\d/gm) ); // 1, 2, 3
```

Without the flag m only the first digit is matched:

```
        let str = `1st place: Winnie
2nd place: Piglet
3rd place: Eeyore`;

console.log( 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.

Please note:
"Start of a line" formally means "immediately after a line break": the test ^ in multiline mode matches at all positions preceded by a newline character \n.
And at the text start.

Searching at line end $The dollar sign $ behaves similarly.
The regular expression \d$ finds the last digit in every line

```
        let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;

console.log( str.match(/\d$/gm) ); // 1,2,3
```

Without the flag m, the dollar $ would only match the end of the whole text, so only the very last digit would be found.

Please note:
"End of a line" formally means "immediately before a line break": the test $ in multiline mode matches at all positions succeeded by a newline character \n.
And at the text end.

Searching for \n instead of ^ $To find a newline, we can use not only anchors ^ and $, but also the newline character \n.
What's the difference? Let's see an example.
Here we search for \d\n instead of \d$:

```
        let str = `Winnie: 1
Piglet: 2
Eeyore: 3`;

console.log( str.match(/\d\n/g) ); // 1\n,2\n
```

As we can see, there are 2 matches instead of 3.

That's because there's no newline after 3 (there's text end though, so it matches $).

Another difference: now every match includes a newline character \n. Unlike the anchors ^ $, that only test the condition (start/end of a line), \n is a character, so it becomes a part of the result.

So, a \n in the pattern is used when we need newline characters in the result, while anchors are used to find something at the beginning/end of a line.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-multiline-mode","identifier":"\/regexp-multiline-mode"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}December 12, 2021Word boundary: \bA word boundary \b is a test, just like ^ and $.

When the regexp engine (program module that implements searching for regexps) comes across \b, it checks that the position in the string is a word boundary.

There are three different positions that qualify as word boundaries:

At string start, if the first string character is a word character \w.

Between two characters in the string, where one is a word character \w and the other is not.

At string end, if the last string character is a word character \w.

For instance, regexp \bJava\b will be found in Hello, Java!, where Java is a standalone word, but not in Hello, JavaScript!.

```
    alert( "Hello, Java!".match(/\bJava\b/) ); // Java
alert( "Hello, JavaScript!".match(/\bJava\b/) ); // null
```

In the string Hello, Java! following positions correspond to \b:

So, it matches the pattern \bHello\b, because:

At the beginning of the string matches the first test \b.
Then matches the word Hello.
Then the test \b matches again, as we're between o and a comma.

So the pattern \bHello\b would match, 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 \w, 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)
```

We can use \b not only with words, but with digits as well.
For example, the pattern \b\d\d\b looks for standalone 2-digit numbers. In other words, it looks for 2-digit numbers that are surrounded by characters different from \w, such as spaces or punctuation (or text start/end).

```
        alert( "1 23 456 78".match(/\b\d\d\b/g) ); // 23,78
alert( "12,34,56".match(/\b\d\d\b/g) ); // 12,34,56
```

Word boundary \b doesn't work for non-latin alphabets

The word boundary test \b checks that there should be \w on the one side from the position and "not \w" – on the other side.

But \w means a latin letter a-z (or a digit or an underscore), so the test doesn't work for other characters, e.g. cyrillic letters or hieroglyphs.

TasksFind the timeThe 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.

solutionThe answer: \b\d\d:\d\d\b.

```
        alert( "Breakfast at 09:00 in the room 123:456.".match( /\b\d\d:\d\d\b/ ) ); // 09:00
```

issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-boundary","identifier":"\/regexp-boundary"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}October 25, 2021Escaping, special charactersAs we've seen, a backslash \ is used to denote character classes, e.g. \d. So it's a special character in regexps (just like in regular strings).

There are other special characters as well, that have special meaning in a regexp, such as [ ] { } ( ) \ ^ $. | ? * +. They are used to do more powerful searches.

Don't try to remember the list – soon we'll deal with each of them, and you'll know them by heart automatically.

EscapingLet's say we want to find literally a dot. 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!)
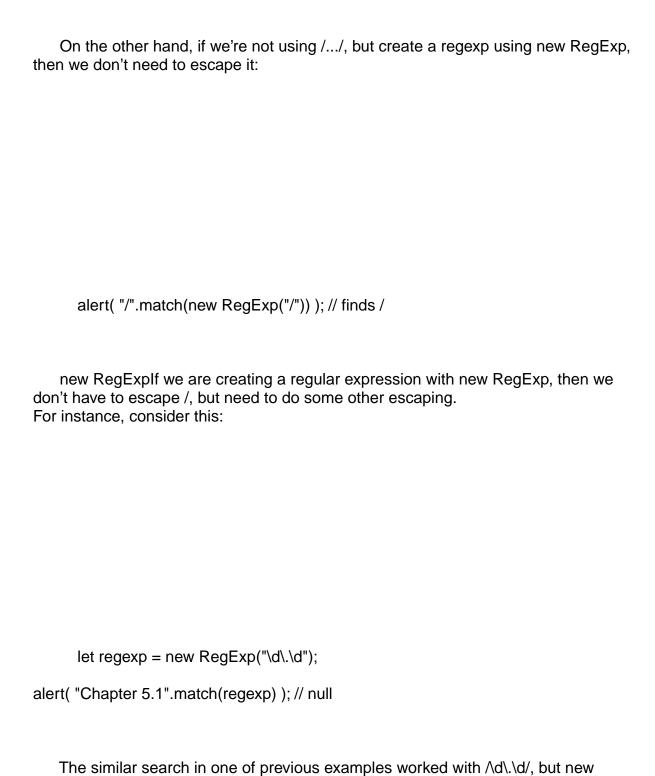alert( "Chapter 511".match(/\d\.\d/) ); // null (looking for a real dot \.)
```

Parentheses are also special characters, so if we want them, we should use \(. The example below looks for a string "g()":

```
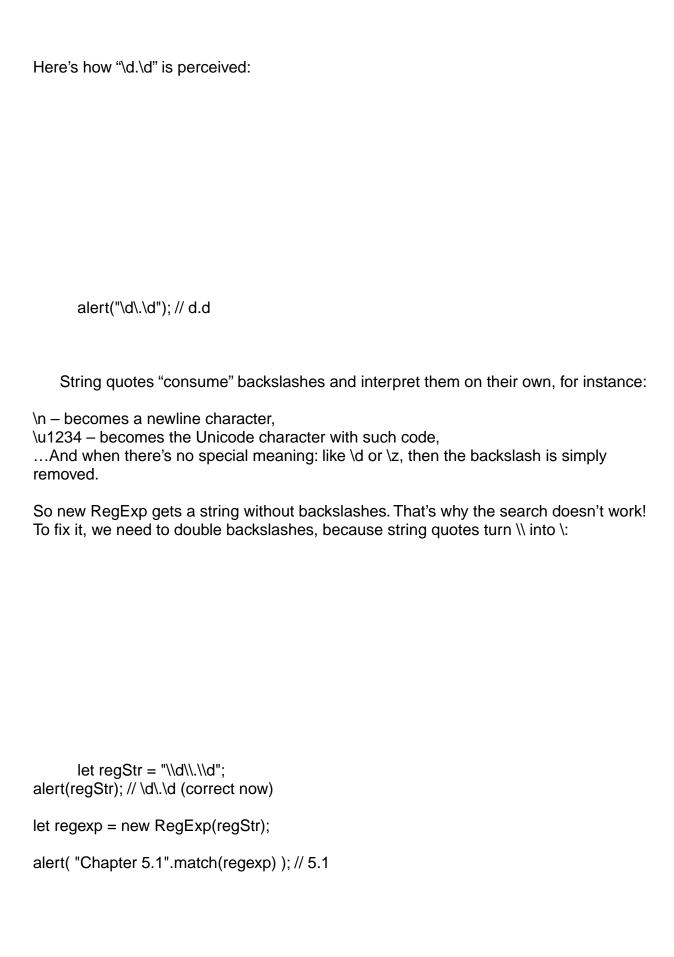alert( "function g()".match(/g\(\)/) ); // "g()"
```

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 slashA 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("/")) ); // finds /
```

new RegExpIf 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 regexp = new RegExp("\d\.\d");

alert( "Chapter 5.1".match(regexp) ); // null
```

The similar search in one of previous examples worked with /\d\.\d/, but new RegExp("\d\.\d") doesn't work, why?
The reason is that backslashes are "consumed" by a string. As we may recall, regular strings have their own special characters, such as \n, and a backslash is used for escaping.

Here's how "\d.\d" is perceived:

```
alert("\d\.\d"); // d.d
```

String quotes "consume" backslashes and interpret them on their own, 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 new RegExp gets a string without backslashes. That's why the search doesn't work!
To fix it, we need to double backslashes, because string quotes turn \\ into \:

```
let regStr = "\\d\\.\\d";
alert(regStr); // \d\.\d (correct now)

let regexp = new RegExp(regStr);

alert( "Chapter 5.1".match(regexp) ); // 5.1
```

Summary

To search for special characters [\ ^ $. | ? * + ( ) literally, we need to prepend them with a backslash \ ("escape them").

We also need to escape / if we're inside /.../ (but not inside new RegExp).

When passing a string to new RegExp, we need to double backslashes \\, cause string quotes consume one of them.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-escaping","identifier":"\/regexp-escaping"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}October 14, 2022Sets and ranges [...]Several characters or character classes inside square brackets […] mean to "search for any character among given".

SetsFor 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 searches for:

V,
then one of the letters [oi],
then la.

So there would be a match for Vola or Vila.
RangesSquare 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
```

Here [0-9A-F] has two ranges: it searches for a character that is either a digit from 0
to 9 or a letter from A to F.
If we'd like to look for lowercase letters as well, we can add the range a-f: [0-9A-Fa-f].

Or add the flag i.
We can also use character classes inside […].
For instance, if we'd like to look for a wordly character \w or a hyphen -, then the set is [\w-].
Combining multiple classes is also possible, e.g. [\s\d] means "a space character or a digit".

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 rare Unicode space characters.

Example: multi-language \wAs the character class \w is a shorthand for [a-zA-Z0-9_], it can't find Chinese hieroglyphs, Cyrillic letters, etc.
We can write a more universal pattern, that looks for wordly characters in any language.
That's easy with Unicode properties: [\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}].
Let's decipher it. Similar to \w, we're making a set of our own that includes characters with following Unicode properties:

Alphabetic (Alpha) – for letters,
Mark (M) – for accents,
Decimal_Number (Nd) – for digits,
Connector_Punctuation (Pc) – for the underscore '_' and similar characters,
Join_Control (Join_C) – two special codes 200c and 200d, used in ligatures, e.g. in Arabic.

An example of use:

```
let regexp = /[\p{Alpha}\p{M}\p{Nd}\p{Pc}\p{Join_C}]/gu;
```

```
let str = `Hi O`Y} 12`;
```

```
// finds all letters and digits:
alert( str.match(regexp) ); // H,i,O`,Y},1,2
```

Of course, we can edit this pattern: add Unicode properties or remove them. Unicode properties are covered in more details in the article Unicode: flag "u" and class \p{...}.

Unicode properties aren't supported in IE

Unicode properties p{…} are not implemented in IE. If we really need them, we can use library XRegExp.

Or just use ranges of characters in a language that interests us, e.g. [C ÔO] for Cyrillic letters.

Excluding rangesBesides 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 .
```

Escaping in […]Usually when we want to find exactly a special character, we need to escape it like \.. And if we need a backslash, then we use \\, and so on.

In square brackets we can use the vast majority of special characters without escaping:

Symbols . + ( ) never need escaping.
A hyphen - is not escaped in the beginning or the end (where it does not define a range).
A caret ^ is only escaped in the beginning (where it means exclusion).
The closing square bracket] is always escaped (if we need to look for that symbol).

In other words, all special characters are allowed without escaping, except when 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 regexp = /[-().^+]/g;

alert( "1 + 2 - 3".match(regexp) ); // Matches +, -
```

…But if you decide to escape them "just in case", then there would be no harm:

```
        // Escaped everything
let regexp = /[\-\(\)\.\^\+]/g;
```

```
alert( "1 + 2 - 3".match(regexp) ); // also works: +, -
```

Ranges and flag "u"If there are surrogate pairs in the set, flag u is required for them to work correctly.
For instance, let's look for [Ø5Ü³Ø5Ü´] in the string Ø5Ü³:

```
alert( 'Ø5Ü³'.match(/[Ø5Ü³Ø5Ü´]/) ); // shows a strange character, like [?]
// (the search was performed incorrectly, half-character returned)
```

The result is incorrect, because by default regular expressions "don't know" about surrogate pairs.
The regular expression engine thinks that [Ø5Ü³Ø5Ü´] – are not two, but four characters:

left half of Ø5Ü³ (1),
right half of Ø5Ü³ (2),
left half of Ø5Ü´ (3),
right half of Ø5Ü´ (4).

We can see their codes like this:

```
      for(let i=0; i<'Ø5Ü³Ø5Ü´'.length; i++) {
  alert('Ø5Ü³Ø5Ü´'.charCodeAt(i)); // 55349, 56499, 55349, 56500
};
```

So, the example above finds and shows the left half of Ø5Ü³.
If we add flag u, then the behavior will be correct:

```
      alert( 'Ø5Ü³'.match(/[Ø5Ü³Ø5Ü´]/u) ); // Ø5Ü³
```

The similar situation occurs when looking for a range, such as [Ø5Ü³-Ø5Ü´].
If we forget to add flag u, there will be an error:

```
      'Ø5Ü³'.match(/[Ø5Ü³-Ø5Ü´]/); // Error: Invalid regular expression
```

The reason is that without flag u surrogate pairs are perceived as two characters,
so [Ø5Ü³-Ø5Ü´] is interpreted as [<55349><56499>-<55349><56500>] (every surrogate pair is
replaced with its codes). Now it's easy to see that the range 56499-55349 is invalid: its
starting code 56499 is greater than the end 55349. That's the formal reason for the
error.

With the flag u the pattern works correctly:

```
        // look for characters from Ø5Ü³ to Ø5Üµ
alert( 'Ø5Ü´'.match(/[Ø5Ü³-Ø5Üµ]/u) ); // Ø5Ü´
```

  TasksJava[^script]We have a regexp /Java[^script]/.
Does it match anything in the string Java? In the string JavaScript?
solutionAnswers: 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 [^script] part matches the character "S". It's not one of script. As the regexp is case-sensitive (no i flag), it treats "S" as a different character from "s".

```
alert( "JavaScript".match(/Java[^script]/) ); // "JavaS"
```

Find the time as hh:mm or hh-mmThe 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 regexp = /your regexp/g;
alert( "Breakfast at 09:00. Dinner at 21-30".match(regexp) ); // 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.
solutionAnswer: \d\d[-:]\d\d.

```
let regexp = /\d\d[-:]\d\d/g;
```

```
alert( "Breakfast at 09:00. Dinner at 21-30".match(regexp) ); // 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.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-character-sets-and-ranges","identifier":"\/regexp-character-sets-and-ranges"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}February 29, 2020Quantifiers +, *, ? 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 can 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
```

ShorthandsThere 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 (may be many or none):

```
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 examplesQuantifiers are used very often. They serve as the main "building block"
of complex regular expressions, so let's see more examples.
Regexp for decimal fractions (a number with a floating point): \d+\.\d+
In action:

```
alert( "0 1 12.345 7890".match(/\d+\.\d+/g) ); // 12.345
```

Regexp for an "opening HTML-tag without attributes", such as <span> or <p>.

The simplest one: /<[a-z]+>/i

```
alert( "<body> ... </body>".match(/<[a-z]+>/gi) ); // <body>
```

The regexp looks for character '<' followed by one or more Latin letters, and then '>'.

Improved: /<[a-z][a-z0-9]*>/i
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>
```

Regexp "opening or closing HTML-tag without attributes": /<\/?[a-z][a-z0-9]*>/i
We added an optional slash /? near the beginning of the pattern. 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>
```

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 as HTML has stricter restrictions for a tag name, <[a-z][a-z0-9]*> is more reliable.

Can we use <\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 remove them from the result by other means.

TasksHow to find an ellipsis "..." ?importance: 5Create a regexp to find ellipsis: 3 (or more?) dots in a row.

Check it:

```
let regexp = /your regexp/g;
alert( "Hello!... How goes?.....".match(regexp) ); // ..., .....
```

solutionSolution:

```
let regexp = /\.{3,}/g;
alert( "Hello!... How goes?.....".match(regexp) ); // ..., .....
```

Please note that the dot is a special character, so we have to escape it and insert as \..

Regexp for HTML colorsCreate a regexp to search HTML-colors written as #ABCDEF: first # and then 6 hexadecimal characters.

An example of use:

```
let regexp = /...your regexp.../

let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2
#12345678";

alert( str.match(regexp) )  // #121212,#AA00ef
```

P.S. In this task we do not need other color formats like #123 or rgb(1,2,3) etc.

solutionWe 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 regexp = /#[a-f0-9]{6}/gi;

let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2"

alert( str.match(regexp) );  // #121212,#AA00ef
```

The problem is that it finds the color in longer sequences:

alert("#12345678".match(/#[a-f0-9]{6}/gi)) // #123456

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
```

codepen…)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-quantifiers","identifier":"\/
regexp-quantifiers"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled =
true;
TutorialRegular expressions{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://
javascript.info/regular-expressions"}]}October 20, 2022Greedy and lazy
quantifiersQuantifiers 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». There exist other quotes,
such as „Witaj, ·v–V6–R @ (Polish) or 0 O`Y}ÿ N uL0 (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 regexp = /".+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // "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 searchTo find a match, the regular expression engine uses the following
algorithm:

For every position in the string

Try to 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 ".+".

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, fails again, and finally finds the quote at the 3rd position:

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:

Then the dot repeats because of the quantifier .+. The regular expression engine adds to the match one character after another.
…Until when? All characters match the dot, so it only stops when it reaches the end of the string:

Now the engine finished repeating .+ 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 string end and tries to match the rest of the pattern from that position.

If there were a quote there, then the search would end, but the last character is 'e', so there's no match.

…So the engine decreases the number of repetitions of .+ by one more character:

The quote '"' does not match 'n'.

The engine keep backtracking: it decreases the count of repetition for '.' until the rest of the pattern (in our case '"') matches:

The match is complete.

So the first match is "witch" and her "broom". If the regular expression has flag g, then the search will continue from where the first match ends. 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) a quantified character is repeated as many times as possible.

The regexp engine adds to the match as many characters as it can for .+, and then shortens that one by one, if the rest of the pattern doesn't match.

For our task we want another thing. That's where a lazy mode can help.

Lazy modeThe lazy mode of quantifiers 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 regexp = /".+?"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // "witch", "broom"
```

To clearly understand the change, let's trace the search step by step.

The first step is the same: it finds the pattern start '"' at the 3rd position:

The next step is also similar: the engine finds a match for the dot '.':

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:

If there were a quote there, then the search would end, but there's 'i', so there's no match.

Then the regular expression engine increases the number of repetitions for the dot and tries one more time:

Failure again. Then the number of repetitions is increased again and again…

…Till the match for the rest of the pattern is found:

The next search starts from the end of the current match and yield one more result:

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+?/) ); // 123 4
```

The pattern \d+ tries to match as many digits as it can (greedy mode), so it finds 123 and stops, because the next character is a space ' '.

Then there's a space in the pattern, it matches.

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.

Optimizations
    Modern regular expression engines can optimize internal algorithms to work faster. So they may work a bit differently 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 approachWith 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 regexp = /"[^"]+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(regexp) ); // "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 <a href="..." class="doc">, with any href.

Which regular expression to use?

The first idea might be: /<a href=".*" class="doc">/g.

Let's check it:

```
let str = '...<a href="link" class="doc">...';
let regexp = /<a href=".*" class="doc">/g;
```

```
// Works!
alert( str.match(regexp) ); // <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 regexp = /<a href=".*" class="doc">/g;

// Whoops! Two links in one match!
alert( str.match(regexp) ); // <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 regexp = /<a href=".*?" class="doc">/g;

// Works!
alert( str.match(regexp) ); // <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 regexp = /<a href=".*?" class="doc">/g;

// Wrong match!
alert( str.match(regexp) ); // <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:

First the regexp finds a link start <a href=".
Then it looks for .*?: takes one character (lazily!), check if there's a match for " class="doc"> (none).
Then takes another character into .*?, and so on… until it finally reaches " class="doc">.

But the problem is: that's already beyond the link <a...>, 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, we need the pattern to look for <a href="...something..." class="doc">, but both greedy and lazy variants have problems.
The correct variant can 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 regexp = /<a href="[^"]*" class="doc">/g;

// Works!
alert( str1.match(regexp) ); // null, no matches, that's correct
alert( str2.match(regexp) ); // <a href="link1" class="doc">, <a href="link2" class="doc">
```

SummaryQuantifiers have two modes of work:

Greedy
By default the regular expression engine tries to repeat the quantified character 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 quantified character.

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, as in the pattern "[^"]+".
TasksA match for /d+? d+?/What's the match here?

```
alert( "123 456".match(/\d+? \d+?/g) ); // ?
```

solutionThe 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.
Find HTML commentsFind all HTML comments in the text:

```
let regexp = /your regexp/g;

let str = `... <!-- My -- comment
 test --> ..  <!----> ..
`;

alert( str.match(regexp) ); // '<!-- My -- comment \n test -->', '<!---->'
```

solutionWe need to find the beginning of the comment <!--, then everything till the end of -->.
An acceptable variant is <!--.*?--> – the lazy quantifier makes the dot stop right before --

>. We also need to add flag s for the dot to include newlines.
Otherwise multiline comments won't be found:

```
let regexp = /<!--.*?-->/gs;

let str = `... <!-- My -- comment
 test --> ..  <!----> ..
`;

alert( str.match(regexp) ); // '<!-- My -- comment \n test -->', '<!---->'
```

Find HTML tagsCreate a regular expression to find all (opening and closing) HTML tags with their attributes.
An example of use:

```
let regexp = /your regexp/g;

let str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(regexp) ); // '<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.
solutionThe solution is <[^<>]+>.

```
let regexp = /<[^<>]+>/g;

let str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(regexp) ); // '<a href="/">', '<input type="radio" checked>', '<b>'
```

ShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:∨∨javascript.info∨regexp-greedy-and-lazy","identifier":"∨regexp-greedy-and-lazy"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]}August 14, 2022Capturing groupsA part of a pattern can be enclosed in parentheses (...). This is called a "capturing group". That has two effects:

It allows to get a part of the match as a separate item in the result array.
If we put a quantifier after the parentheses, it applies to the parentheses as a whole.

ExamplesLet's see how parentheses work in examples.
Example: gogogoWithout parentheses, the pattern go+ means g character, followed by

o repeated one or more times. For instance, goooo or gooooooooo.
Parentheses group characters together, so (go)+ means go, gogo, gogogo and so on.

```
alert( 'Gogogo now!'.match(/(go)+/ig) ); // "Gogogo"
```

Example: domainLet's make something more complex – a regular expression to search for a website domain.
For example:

```
mail.com
users.mail.com
smith.users.mail.com
```

As we can see, a domain consists of repeated words, a dot after each one except the last one.
In regular expressions that's (\w+\.)+\w+:

```
let regexp = /(\w+\.)+\w+/g;
```

alert("site.com my.site.com".match(regexp)); // site.com,my.site.com

The search works, but the pattern can't match a domain with a hyphen, e.g. my-site.com, because the hyphen does not belong to class \w.

We can fix it by replacing \w with [\w-] in every word except the last one: ([\w-]+\.)+\w+.

Example: emailThe previous example can be extended. We can create a regular expression for emails based on it.

The email format is: name@domain. Any word can be the name, hyphens and dots are allowed. In regular expressions that's [-.\w]+.

The pattern:

```
let regexp = /[-.\w]+@([\w-]+\.)+[\w-]+/g;
```

alert("my@mail.com @ his@site.com.uk".match(regexp)); // my@mail.com, his@site.com.uk

That regexp is not perfect, but mostly works and helps to fix accidental mistypes. The only truly reliable check for an email can only be done by sending a letter.

Parentheses contents in the matchParentheses are numbered from left to right. The search engine memorizes the content matched by each of them and allows to get it in the result.

The method str.match(regexp), if regexp has no flag g, looks for the first match and returns it as an array:

At index 0: the full match.
At index 1: the contents of the first parentheses.
At index 2: the contents of the second parentheses.
…and so on…

For instance, we'd like to find HTML tags <.*?>, and process them. It would be convenient to have tag content (what's inside the angles), in a separate variable.
Let's wrap the inner content into parentheses, like this: <(.*?)>.
Now we'll get both the tag as a whole <h1> and its contents h1 in the resulting array:

```
let str = '<h1>Hello, world!</h1>';

let tag = str.match(/<(.*?)>/);

alert( tag[0] ); // <h1>
alert( tag[1] ); // h1
```

Nested groupsParentheses can be nested. In this case the numbering also goes from left to right.
For instance, when searching a tag in <span class="my"> we may be interested in:

The tag content as a whole: span class="my".
The tag name: span.
The tag attributes: class="my".

Let's add parentheses for them: <(([a-z]+)\s*([^>]*))>.
Here's how they are numbered (left to right, by the opening paren):

In action:

```
        let str = '<span class="my">';

let regexp = /<(([a-z]+)\s*([^>]*))>/;

let result = str.match(regexp);
alert(result[0]); // <span class="my">
alert(result[1]); // span class="my"
alert(result[2]); // span
alert(result[3]); // class="my"
```

The zero index of result always holds the full match.
Then groups, numbered from left to right by an opening paren. The first group is
returned as result[1]. Here it encloses the whole tag content.
Then in result[2] goes the group from the second opening paren ([a-z]+) – tag name,
then in result[3] the tag: ([^>]*).
The contents of every group in the string:

Optional groupsEven if a group is optional and doesn't exist in the match (e.g. has
the quantifier (...)?), the corresponding result array item 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 ac:

```
        let match = 'ac'.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"].
Searching for all matches with groups: matchAll
       matchAll is a new method, polyfill may be needed
       The method matchAll is not supported in old browsers.
A polyfill may be required, such as https://github.com/ljharb/String.prototype.matchAll.

When we search for all matches (flag g), the match method does not return contents for
groups.
For example, let's find all tags in a string:

```
let str = '<h1> <h2>';

let tags = str.match(/<(.*?)>/g);

alert( tags ); // <h1>,<h2>
```

The result is an array of matches, but without details about each of them. But in
practice we usually need contents of capturing groups in the result.
To get them, we should search using the method str.matchAll(regexp).
It was added to JavaScript language long after match, as its "new and improved
version".
Just like match, it looks for matches, but there are 3 differences:

It returns not an array, but an iterable object.
When the flag g is present, it returns every match as an array with groups.
If there are no matches, it returns not null, but an empty iterable object.

For instance:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

// results - is not an array, but an iterable object
alert(results); // [object RegExp String Iterator]
```

```
alert(results[0]); // undefined (*)

results = Array.from(results); // let's turn it into array

alert(results[0]); // <h1>,h1 (1st tag)
alert(results[1]); // <h2>,h2 (2nd tag)
```

As we can see, the first difference is very important, as demonstrated in the line (*). We can't get the match as results[0], because that object isn't pseudoarray. We can turn it into a real Array using Array.from. There are more details about pseudoarrays and iterables in the article Iterables.
There's no need in Array.from if we're looping over results:

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

for(let result of results) {
  alert(result);
  // first alert: <h1>,h1
  // second: <h2>,h2
}
```

…Or using destructuring:

```
let [tag1, tag2] = '<h1> <h2>'.matchAll(/<(.*?)>/gi);
```

Every match, returned by matchAll, has the same format as returned by match without flag g: it's an array with additional properties index (match index in the string) and input (source string):

```
let results = '<h1> <h2>'.matchAll(/<(.*?)>/gi);

let [tag1, tag2] = results;

alert( tag1[0] ); // <h1>
alert( tag1[1] ); // h1
alert( tag1.index ); // 0
alert( tag1.input ); // <h1> <h2>
```

Why is a result of matchAll an iterable object, not an array?
Why is the method designed like that? The reason is simple – for the optimization.
The call to matchAll does not perform the search. Instead, it returns an iterable object, without the results initially. The search is performed each time we iterate over it, e.g. in the loop.
So, there will be found as many results as needed, not more.
E.g. there are potentially 100 matches in the text, but in a for..of loop we found 5 of them, then decided it's enough and made a break. Then the engine won't spend time finding other 95 matches.

Named groupsRemembering groups by their numbers is hard. For simple patterns it's doable, but for more complex ones counting parentheses is inconvenient. We have a much better option: give names to parentheses.
That's done by putting ?<name> immediately after the opening paren.
For example, let's look for a date in the format "year-month-day":

```
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.
To look for all dates, we can add flag g.
We'll also need matchAll to obtain full matches, together with groups:

```
let dateRegexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30 2020-01-01";

let results = str.matchAll(dateRegexp);

for(let result of results) {
  let {year, month, day} = result.groups;

  alert(`${day}.${month}.${year}`);
  // first alert: 30.10.2019
```

```
  // second: 01.01.2020
}
```

Capturing groups in replacementMethod str.replace(regexp, replacement) that replaces all matches with regexp in str allows to use parentheses contents in the replacement string. That's done using $n, where n is the group number.
For example,

```
    let str = "John Bull";
let regexp = /(\w+) (\w+)/;

alert( str.replace(regexp, '$2, $1') ); // Bull, John
```

For named parentheses the reference will be $<name>.
For example, let's reformat dates from "year-month-day" to "day.month.year":

```
    let regexp = /(?<year>[0-9]{4})-(?<month>[0-9]{2})-(?<day>[0-9]{2})/g;

let str = "2019-10-30, 2020-01-01";
```

```
alert( str.replace(regexp, '$<day>.$<month>.$<year>') );
// 30.10.2019, 01.01.2020
```

Non-capturing groups with ?:

Sometimes we need parentheses to correctly apply a quantifier, but we don't want their 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 the parentheses contents (go) as a separate array item, we can write: (?:go)+.

In the example below we only get the name John as a separate member of the match:

```
let str = "Gogogo John!";

// ?: excludes 'go' from capturing
let regexp = /(?:go)+ (\w+)/i;

let result = str.match(regexp);

alert( result[0] ); // Gogogo John (full match)
alert( result[1] ); // John
alert( result.length ); // 2 (no more items in the array)
```

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 obtained in the results:

The method str.match returns capturing groups only without flag g.
The method str.matchAll always returns capturing groups.

If the parentheses have no name, then their contents is available in the match array by

its number. Named parentheses are also available in the property groups.

We can also use parentheses contents in the replacement string in str.replace: by the number $n or the name $<name>.

A group may be excluded from numbering by adding ?: in its start. That's used when we need to apply a quantifier to the whole group, but don't want it as a separate item in the results array. We also can't reference such parentheses in the replacement string.

TasksCheck MAC-addressMAC-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 regexp = /your regexp/;

alert( regexp.test('01:32:54:67:89:AB') ); // true

alert( regexp.test('0132546789AB') ); // false (no colons)

alert( regexp.test('01:32:54:67:89') ); // false (5 numbers, must be 6)

alert( regexp.test('01:32:54:67:89:ZZ') ) // false (ZZ at the end)
```

solutionA two-digit hex number is [0-9a-f]{2} (assuming the flag i is set).

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 regexp = /^[0-9a-f]{2}(:[0-9a-f]{2}){5}$/i;
```

```
alert( regexp.test('01:32:54:67:89:AB') ); // true

alert( regexp.test('0132546789AB') ); // false (no colons)

alert( regexp.test('01:32:54:67:89') ); // false (5 numbers, need 6)

alert( regexp.test('01:32:54:67:89:ZZ') ) // false (ZZ in the end)
```

Find color in the format #abc or #abcdefWrite a RegExp that matches colors in the format #abc or #abcdef. That is: # followed by 3 or 6 hexadecimal digits.
Usage example:

```
let regexp = /your regexp/g;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(regexp) ); // #3f3 #AA00ef
```

P.S. This should be exactly 3 or 6 hex digits. Values with 4 digits, such as #abcd, should not match.
solutionA 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. The color has either 3 or 6 digits.
Let's use the quantifier {1,2} for that: we'll have /#([a-f0-9]{3}){1,2}/i.
Here the pattern [a-f0-9]{3} is enclosed in parentheses to apply the quantifier {1,2}.
In action:

```
let regexp = /#([a-f0-9]{3}){1,2}/gi;
```

```
let str = "color: #3f3; background-color: #AA00ef; and: #abcd";
```

```
alert( str.match(regexp) ); // #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 regexp = /#([a-f0-9]{3}){1,2}\b/gi;
```

```
let str = "color: #3f3; background-color: #AA00ef; and: #abcd";
```

```
alert( str.match(regexp) ); // #3f3 #AA00ef
```

Find all numbersWrite a regexp that looks for all decimal numbers including integer ones, with the floating point and negative ones.
An example of use:

```
let regexp = /your regexp/g;
```

```
let str = "-1.5 0 2 -123.4.";
```

```
alert( str.match(regexp) ); // -1.5, 0, 2, -123.4
```

solutionA positive number with an optional decimal part is: \d+(\.\d+)?.

Let's add the optional - in the beginning:

```
let regexp = /-?\d+(\.\d+)?/g;

let str = "-1.5 0 2 -123.4.";

alert( str.match(regexp) );   // -1.5, 0, 2, -123.4
```

Parse an expressionAn 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:

The first number.
The operator.
The second number.

For example:

```
let [a, op, b] = parse("1.2 * 3.4");

alert(a); // 1.2
alert(op); // *
```

alert(b); // 3.4

solutionA regexp for a number is: -?\d+(\.\d+)?. We created it in the previous task.
An operator is [-+*/]. The hyphen - goes first in the square brackets, because in the
middle it would mean a character range, while we just want a character -.
The 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+)?.
It has 3 parts, with \s* between them:

-?\d+(\.\d+)? – the first number,
[-+*/] – the operator,
-?\d+(\.\d+)? – the second number.

To make each of these parts a separate element of the result array, let's enclose them
in parentheses: (-?\d+(\.\d+)?)\s*([-+*/])\s*(-?\d+(\.\d+)?).
In action:

let regexp = /(-?\d+(\.\d+)?)\s*([-+*V/])\s*(-?\d+(\.\d+)?)/;

alert("1.2 + 12".match(regexp));

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 ([-+*V]) – 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, so let's "clean" the result a bit.
The full match (the arrays first item) can be removed by shifting the array result.shift().
Groups that contain decimal parts (number 2 and 4) (.\d+) can be excluded by adding ?: to the beginning: (?:\.\d+)?.
The final solution:

```
      function parse(expr) {
  let regexp = /(-?\d+(?:\.\d+)?)\s*([-+*\/])\s*(-?\d+(?:\.\d+)?)/;

  let result = expr.match(regexp);

  if (!result) return [];
  result.shift();

  return result;
}

alert( parse("-1.23 * 3.45") );  // -1.23, *, 3.45
```

As an alternative to using the non-capturing ?:, we could name the groups, like this:

```
    function parse(expr) {
  let regexp = /(?<a>-?\d+(?:\.\d+)?)\s*(?<operator>[-+*V])\s*(?<b>-?\d+(?:\.\d+)?)/;

  let result = expr.match(regexp);

  return [result.groups.a, result.groups.operator, result.groups.b];
}

alert( parse("-1.23 * 3.45") );  // -1.23, *, 3.45;
```

Backreferences in pattern: \N and \k<name>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: \NA group can be referenced in the pattern using \N, where N is the group number.
To make clear why that's helpful, let's consider a task.
We need to find quoted strings: either single-quoted '...' or a double-quoted "..." – both variants should match.
How to find 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 in the string "She's the one!":

```
let str = `He said: "She's the one!".`;
```

```
let regexp = /['"](.*?)['"]/g;
```

```
// The result is not what we'd like to have
alert( str.match(regexp) ); // "She'
```

As we can see, the pattern found an opening quote ", then the text is consumed 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 backreference it: (['"])(.*?)\1.
Here's the correct code:

```
let str = `He said: "She's the one!".`;
```

```
let regexp = /(['"])(.*?)\1/g;
```

```
alert( str.match(regexp) ); // "She's the one!"
```

Now it works! The regular expression engine finds the first quote (['"]) and memorizes its content. 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.
Similar to that, \2 would mean the contents of the second group, \3 – the 3rd group, and so on.

Please note:

If we use ?: in the group, then we can't reference it. Groups that are excluded from capturing (?:...) are not memorized by the engine.

Don't mess up: in the pattern \1, in the replacement: $1

In the replacement string we use a dollar sign: $1, while in the pattern – a backslash \1.

Backreference by name: \k<name>If a regexp has many parentheses, it's convenient to give them names.
To reference a named group we can use \k<name>.
In the example below the group with quotes is named ?<quote>, so the backreference is \k<quote>:

```
let str = `He said: "She's the one!".`;

let regexp = /(?<quote>['"])(.*?)\k<quote>/g;

alert( str.match(regexp) ); // "She's the one!"
```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-backreferences","identifier":"\/regexp-backreferences"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;
TutorialRegular expressions{"@context":"https://schema.org","@type":"BreadcrumbList","itemListElement":[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},

{"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://javascript.info/regular-expressions"}]]}August 7, 2022Alternation (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 regexp = /html|php|css|java(script)?/gi;

let str = "First HTML appeared, then CSS, then JavaScript";

alert( str.match(regexp) ); // 'HTML', 'CSS', 'JavaScript'
```

We already saw a similar thing – square brackets. They allow to choose between multiple characters, for instance gr[ae]y matches gray or grey.

Square brackets allow only characters or character classes. 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.
gra|ey means gra or ey.

To apply alternation to a chosen part of the pattern, we can enclose it in parentheses:

I love HTML|CSS matches I love HTML or CSS.
I love (HTML|CSS) matches I love HTML or I love CSS.

Example: regexp for timeIn previous articles 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 minutes match the pattern, but that time is invalid).

How can we make a better pattern?

We can use more careful matching. First, the hours:

If the first digit is 0 or 1, then the next digit can be any: [01]\d.
Otherwise, if the first digit is 2, then the next must be [0-3].
(no other first digit is allowed)

We can write both variants in a regexp using alternation: [01]\d|2[0-3].
Next, minutes must be from 00 to 59. In the regular expression language that can be written as [0-5]\d: the first digit 0-5, and then any digit.
If we glue hours and minutes together, we get 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 is: minutes are added to the second alternation variant, here's a clear picture:

[01]\d | 2[0-3]:[0-5]\d

That pattern looks for [01]\d or 2[0-3]:[0-5]\d.
But that's wrong, the alternation should only be used in the "hours" part of the regular expression, to allow [01]\d OR 2[0-3]. Let's correct that by enclosing "hours" into parentheses: ([01]\d|2[0-3]):[0-5]\d.
The final solution:

```
let regexp = /([01]\d|2[0-3]):[0-5]\d/g;

alert("00:00 10:10 23:59 25:99 1:2".match(regexp)); // 00:00,10:10,23:59
```

TasksFind programming languagesThere 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 regexp = /your regexp/g;
```

alert("Java JavaScript PHP C++ C".match(regexp)); // Java JavaScript PHP C++ C

solutionThe first idea can be to list the languages with | in-between.
But that doesn't work right:

```
let regexp = /Java|JavaScript|PHP|C|C\+\+/g;
```

let str = "Java, JavaScript, PHP, C, C++";

alert(str.match(regexp)); // 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:

Change the order to check the longer match first: JavaScript|Java|C\+\+|C|PHP.
Merge variants with the same start: Java(Script)?|C(\+\+)?|PHP.

In action:

```
let regexp = /Java(Script)?|C(\+\+)?|PHP/g;
```

```
let str = "Java, JavaScript, PHP, C, C++";
```

```
alert( str.match(regexp) ); // Java,JavaScript,PHP,C,C++
```

Find bbtag pairsA "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]

Can't happen:
[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 regexp = /your regexp/flags;

let str = "..[url]http://google.com[/url]..";
alert( str.match(regexp) ); // [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 regexp = /your regexp/flags;

let str = "..[url][b]http://google.com[/b][/url]..";
alert( str.match(regexp) ); // [url][b]http://google.com[/b][/url]
```

solutionOpening 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 regexp = /\[(b|url|quote)].*?\[\/\1]/gs;

let str = `
  [b]hello![/b]
  [quote]
    [url]http://google.com[/url]
  [/quote]
`;

alert( str.match(regexp) ); // [b]hello![/b],[quote][url]http://google.com[/url][/quote]
```

Please note that besides escaping [, we had to escape a slash for the closing tag [\/\1], because normally the slash closes the pattern.
Find quoted stringsCreate 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 backslash 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 backslash inside)
.. "\\ \"" ..  (double backslash and an escaped quote inside)
```

In JavaScript we need to double the backslashes 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\"!" .. "\\ \"" ..
```

solutionThe solution: /"(\\.|[^"\\])*"/g.
Step by step:

First we look for an opening quote "
Then if we have a backslash \\ (we have to double it in the pattern because it is a special character), 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 regexp = /"(\\.|[^"\\])*"/g;
let str = ' .. "test me" .. "Say \\"Hello\\"!" .. "\\\\ \\"" .. ';
```

```
alert( str.match(regexp) ); // "test me","Say \"Hello\"!","\\ \""
```

Find the full tagWrite 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 regexp = /your regexp/g;

alert( '<style> <styler> <style test="...">'.match(regexp) ); // <style>, <style test="...">
```

solutionThe 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 regexp = /<style(>|\s.*?>)/g;

alert( '<style> <styler> <style test="...">'.match(regexp) ); // <style>, <style test="...">
```

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several

lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:VVjavascript.infoVregexp-alternation","identifier":"V regexp-alternation"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;

TutorialRegular expressions{"@context":"https:// schema.org","@type":"BreadcrumbList","itemListElement": [{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"}, {"@type":"ListItem","position":2,"name":"Regular expressions","item":"https:// javascript.info/regular-expressions"}]}June 7, 2022Lookahead and lookbehindSometimes we need to find only those matches for a pattern that are followed or preceded by another pattern.

There's a special syntax for that, called "lookahead" and "lookbehind", together referred to as "lookaround".

For the start, let's find the price from the string like 1 turkey costs 30€. That is: a number, followed by € sign.

LookaheadThe syntax is: X(?=Y), it means "look for X, but match only if followed by Y". There may be any pattern instead of X and Y.

For an integer number followed by €, the regexp will be \d+(?=€):

```
let str = "1 turkey costs 30€";

alert( str.match(/\d+(?=€)/) ); // 30, the number 1 is ignored, as it's not followed by €
```

Please note: the lookahead is merely a test, the contents of the parentheses (?=...) is not included in the result 30.

When we look for X(?=Y), the regular expression engine finds X and then checks if there's Y immediately after it. If it's not so, then the potential match is skipped, and the search continues.

More complex tests are possible, e.g. X(?=Y)(?=Z) means:

Find X.

Check if Y is immediately after X (skip if isn't).

Check if Z is also immediately after X (skip if isn't).
If both tests passed, then the X is a match, otherwise continue searching.

In other words, such pattern means that we're looking for X followed by Y and Z at the same time.
That's only possible if patterns Y and Z aren't mutually exclusive.
For example, \d+(?=\s)(?=.*30) looks for \d+ that is followed by a space (?=\s), and there's 30 somewhere after it (?=.*30):

```
let str = "1 turkey costs 30€";

alert( str.match(/\d+(?=\s)(?=.*30)/) ); // 1
```

In our string that exactly matches the number 1.
Negative lookaheadLet's say that we want a quantity instead, not a price from the same string. That's a number \d+, NOT followed by €.
For that, 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+\b(?!€)/g) ); // 2 (the price is not matched)
```

Lookbehind
    Lookbehind browser compatibility
    Please Note: Lookbehind is not supported in non-V8 browsers, such as Safari,
Internet Explorer.

Lookahead allows to add a condition for "what follows".
Lookbehind is similar, but it looks behind. That is, it allows to match a pattern only if
there's something before it.
The syntax is:

Positive lookbehind: (?<=Y)X, matches X, but only if there's Y before it.
Negative lookbehind: (?<!Y)X, matches X, but only if there's no Y before it.

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";

// the dollar sign is escaped \$
alert(str.match(/(?<=\$)\d+/)); // 30 (skipped the sole number)

    And, if we need the quantity – a number, not preceded by $, then we can use a
negative lookbehind (?<!\$)\d+:

```
let str = "2 turkeys cost $60";
```

```
alert( str.match(/(?<!\$)\b\d+/g) ); // 2 (the price is not matched)
```

Capturing groupsGenerally, the contents inside lookaround parentheses does not become a part of the result.

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 lookaround expression as well, or a part of it. That's possible. Just wrap that part into additional parentheses.

In the example below the currency sign (€|kr) is captured, along with the amount:

```
let str = "1 turkey costs 30€";
let regexp = /\d+(?=(€|kr))/; // extra parentheses around €|kr
```

```
alert( str.match(regexp) ); // 30, €
```

And here's the same for lookbehind:

```
let str = "1 turkey costs $30";
let regexp = /(?<=(\$|£))\d+/;

alert( str.match(regexp) ); // 30, $
```

SummaryLookahead and lookbehind (commonly referred to as "lookaround") are useful when we'd like to match something 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.match (without flag g) and str.matchAll (always) return matches as arrays with index property, so we know where exactly in the text it is, and can check the context.
But generally lookaround is more convenient.
Lookaround types:

Pattern
type
matches

X(?=Y)
Positive lookahead
X if followed by Y

X(?!Y)
Negative lookahead
X if not followed by Y

(?<=Y)X
Positive lookbehind
X if after Y

(?<!Y)X
Negative lookbehind
X if not after Y

TasksFind non-negative integersThere's a string of integer numbers.
Create a regexp that looks for only non-negative ones (zero is allowed).
An example of use:

```
let regexp = /your regexp/g;

let str = "0 12 -5 123 -18";

alert( str.match(regexp) ); // 0, 12, 123
```

solutionThe regexp for an integer number is \d+.
We can exclude negatives by prepending it with the negative lookbehind: (?<!-)\d+.
Although, if we try it now, we may notice one more "extra" result:

```
let regexp = /(?<!-)\d+/g;

let str = "0 12 -5 123 -18";

console.log( str.match(regexp) ); // 0, 12, 123, 8
```

As you can see, it matches 8, from -18. To exclude it, we need to ensure that the regexp starts matching a number not from the middle of another (non-matching) number.

We can do it by specifying another negative lookbehind: (?<!-)(?<!\d)\d+. Now (?<!\d) ensures that a match does not start after another digit, just what we need.
We can also join them into a single lookbehind here:

```
let regexp = /(?<![-\d])\d+/g;

let str = "0 12 -5 123 -18";

alert( str.match(regexp) ); // 0, 12, 123
```

Insert After HeadWe have a string with an HTML Document.
Write a regular expression that inserts <h1>Hello</h1> immediately after <body> tag. The tag may have attributes.
For instance:

```
let regexp = /your regular expression/;

let str = `
<html>
  <body style="height: 200px">
  ...
  </body>
</html>
`;

str = str.replace(regexp, `<h1>Hello</h1>`);
```

After that the value of str should be:

```
  <html>
<body style="height: 200px"><h1>Hello</h1>
...
 </body>
</html>
```

solutionIn order to insert after the <body> tag, we must first find it. We can use the regular expression pattern <body.*?> for that.

In this task, we don't need to modify the <body> tag. We only need to add the text after it.

Here's how we can do it:

```
  let str = '...<body style="...">...';
str = str.replace(/<body.*?>/, '$&<h1>Hello</h1>');

alert(str); // ...<body style="..."><h1>Hello</h1>...
```

In the replacement string $& means the match itself, that is, the part of the source text that corresponds to <body.*?>. It gets replaced by itself plus <h1>Hello</h1>.

An alternative is to use lookbehind:

```
    let str = '...<body style="...">...';
str = str.replace(/(?<=<body.*?>)/, `<h1>Hello</h1>`);

alert(str); // ...<body style="..."><h1>Hello</h1>...
```

As you can see, there's only lookbehind part in this regexp.
It works like this:

At every position in the text.
Check if it's preceded by <body.*?>.
If it's so, then we have the match.

The tag <body.*?> won't be returned. The result of this regexp is literally an empty
string, but it matches only at positions preceded by <body.*?>.
So it replaces the "empty line", preceded by <body.*?>, with <h1>Hello</h1>. That's the
insertion after <body>.
P.S. Regexp flags, such as s and i can also be useful: /<body.*?>/si. The s flag makes
the dot . match a newline character, and i flag makes <body> also match <BODY> case-
insensitively.
Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before
commenting…If you have suggestions what to improve - please submit a GitHub issue
or a pull request instead of commenting.If you can't understand something in the article
– please elaborate.To insert few words of code, use the <code> tag, for several lines –
wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin,
codepen…)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-lookahead-
lookbehind","identifier":"\/regexp-lookahead-lookbehind"}); };var disqus_shortname =
"javascriptinfo";var disqus_enabled = true;
TutorialRegular expressions{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://
javascript.info/regular-expressions"}]}October 14, 2022Catastrophic backtrackingSome
regular expressions are looking simple, but can execute a veeeeeery long time, and
even "hang" the JavaScript engine.
Sooner or later most developers occasionally face such behavior. The typical symptom
– a regular expression works fine sometimes, but for certain strings it "hangs",
consuming 100% of CPU.
In such case a web-browser suggests to kill the script and reload the page. Not a good

thing for sure.

For server-side JavaScript such a regexp may hang the server process, that's even worse. So we definitely should take a look at it.

ExampleLet's say we have a string, and we'd like to check if it consists of words \w+ with an optional space \s? after each.

An obvious way to construct a regexp would be to take a word followed by an optional space \w+\s? and then repeat it with *.

That leads us to the regexp ^(\w+\s?)*$, it specifies zero or more such words, that start at the beginning ^ and finish at the end $ of the line.

In action:

```
let regexp = /^(\w+\s?)*$/;

alert( regexp.test("A good string") ); // true
alert( regexp.test("Bad characters: $@#") ); // false
```

The regexp seems to work. The result is correct. Although, on certain strings it takes a lot of time. So long that JavaScript engine "hangs" with 100% CPU consumption.

If you run the example below, you probably won't see anything, as JavaScript will just "hang". A web-browser will stop reacting on events, the UI will stop working (most browsers allow only scrolling). After some time it will suggest to reload the page. So be careful with this:

```
        let regexp = /^(\w+\s?)*$/;
let str = "An input string that takes a long time or even makes this regexp hang!";

// will take a very long time
alert( regexp.test(str) );
```

To be fair, let's note that some regular expression engines can handle such a search effectively, for example V8 engine version starting from 8.8 can do that (so Google Chrome 88 doesn't hang here), while Firefox browser does hang.
Simplified exampleWhat's the matter? Why does the regular expression hang?
To understand that, let's simplify the example: remove spaces \s?. Then it becomes ^(\w+)*$.
And, to make things more obvious, let's replace \w with \d. The resulting regular expression still hangs, for instance:

```
        let regexp = /^(\d+)*$/;

let str = "012345678901234567890123456789z";

// will take a very long time (careful!)
alert( regexp.test(str) );
```

So what's wrong with the regexp?
First, one may notice that the regexp (\d+)* is a little bit strange. The quantifier * looks extraneous. If we want a number, we can use \d+.
Indeed, the regexp is artificial; we got it by simplifying the previous example. But the reason why it is slow is the same. So let's understand it, and then the previous example will become obvious.
What happens during the search of ^(\d+)*$ in the line 123456789z (shortened a bit for clarity, please note a non-digit character z at the end, it's important), why does it take so long?

Here's what the regexp engine does:

First, the regexp engine tries to find the content of the parentheses: the number \d+.
The plus + is greedy by default, so it consumes all digits:

```
       \d+.......
(123456789)z
```

After all digits are consumed, \d+ is considered found (as 123456789).
Then the star quantifier (\d+)* applies. But there are no more digits in the text, so the star doesn't give anything.
The next character in the pattern is the string end $. But in the text we have z instead, so there's no match:

```
               X
\d+........$
(123456789)z
```

As there's no match, the greedy quantifier + decreases the count of repetitions, backtracks one character back.
Now \d+ takes all digits except the last one (12345678):

```
       \d+.......
(12345678)9z
```

Then the engine tries to continue the search from the next position (right after 12345678).

The star (\d+)* can be applied – it gives one more match of \d+, the number 9:

```
        \d+.......\d+
(12345678)(9)z
```

The engine tries to match $ again, but fails, because it meets z instead:

```
                X
\d+.......\d+
(12345678)(9)z
```

There's no match, so the engine will continue backtracking, decreasing the number of repetitions. Backtracking generally works like this: the last greedy quantifier decreases the number of repetitions until it reaches the minimum. Then the previous greedy quantifier decreases, and so on.
All possible combinations are attempted. Here are their examples.
The first number \d+ has 7 digits, and then a number of 2 digits:

```
                X
\d+......\d+
(1234567)(89)z
```

The first number has 7 digits, and then two numbers of 1 digit each:

```
            X
\d+......\d+\d+
```

(1234567)(8)(9)z

The first number has 6 digits, and then a number of 3 digits:

            X
\d+.......\d+
(123456)(789)z

The first number has 6 digits, and then 2 numbers:

            X
\d+.....\d+ \d+
(123456)(78)(9)z

…And so on.

There are many ways to split a sequence of digits 123456789 into numbers. To be precise, there are 2n-1, where n is the length of the sequence.

For 123456789 we have n=9, that gives 511 combinations.
For a longer sequence with n=20 there are about one million (1048575) combinations.
For n=30 – a thousand times more (1073741823 combinations).

Trying each of them is exactly the reason why the search takes so long.
Back to words and stringsThe similar thing happens in our first example, when we look for words by pattern ^(\w+\s?)*$ in the string An input that hangs!.
The reason is that a word can be represented as one \w+ or many:

      (input)
(inpu)(t)

(inp)(u)(t)
(in)(p)(ut)
...

For a human, it's obvious that there may be no match, because the string ends with an exclamation sign !, but the regular expression expects a wordly character \w or a space \s at the end. But the engine doesn't know that.

It tries all combinations of how the regexp (\w+\s?)* can "consume" the string, including variants with spaces (\w+\s)* and without them (\w+)* (because spaces \s? are optional). As there are many such combinations (we've seen it with digits), the search takes a lot of time.

What to do?

Should we turn on the lazy mode?

Unfortunately, that won't help: if we replace \w+ with \w+?, the regexp will still hang. The order of combinations will change, but not their total count.

Some regular expression engines have tricky tests and finite automations that allow to avoid going through all combinations or make it much faster, but most engines don't, and it doesn't always help.

How to fix?There are two main approaches to fixing the problem.

The first is to lower the number of possible combinations.

Let's make the space non-optional by rewriting the regular expression as ^(\w+\s)*\w*$ – we'll look for any number of words followed by a space (\w+\s)*, and then (optionally) a final word \w*.

This regexp is equivalent to the previous one (matches the same) and works well:

```
let regexp = /^(\w+\s)*\w*$/;
let str = "An input string that takes a long time or even makes this regex hang!";

alert( regexp.test(str) ); // false
```


Why did the problem disappear?

That's because now the space is mandatory.

The previous regexp, if we omit the space, becomes (\w+)*, leading to many combinations of \w+ within a single word

So input could be matched as two repetitions of \w+, like this:

```
    \w+  \w+
(inp)(ut)
```

The new pattern is different: (\w+\s)* specifies repetitions of words followed by a space! The input string can't be matched as two repetitions of \w+\s, because the space is mandatory.

The time needed to try a lot of (actually most of) combinations is now saved.

Preventing backtrackingIt's not always convenient to rewrite a regexp though. In the example above it was easy, but it's not always obvious how to do it.

Besides, a rewritten regexp is usually more complex, and that's not good. Regexps are complex enough without extra efforts.

Luckily, there's an alternative approach. We can forbid backtracking for the quantifier.

The root of the problem is that the regexp engine tries many combinations that are obviously wrong for a human.

E.g. in the regexp (\d+)*$ it's obvious for a human, that + shouldn't backtrack. If we replace one \d+ with two separate \d+\d+, nothing changes:

```
    \d+........
(123456789)!

\d+...\d+....
(1234)(56789)!
```

And in the original example ^(\w+\s?)*$ we may want to forbid backtracking in \w+. That is: \w+ should match a whole word, with the maximal possible length. There's no need to lower the repetitions count in \w+ or to split it into two words \w+\w+ and so on.

Modern regular expression engines support possessive quantifiers for that. Regular quantifiers become possessive if we add + after them. That is, we use \d++ instead of \d+ to stop + from backtracking.

Possessive quantifiers are in fact simpler than "regular" ones. They just match as many as they can, without any backtracking. The search process without backtracking is simpler.

There are also so-called "atomic capturing groups" – a way to disable backtracking inside parentheses.
…But the bad news is that, unfortunately, in JavaScript they are not supported.
We can emulate them though using a "lookahead transform".
Lookahead to the rescue!So we've come to real advanced topics. We'd like a quantifier, such as + not to backtrack, because sometimes backtracking makes no sense.
The pattern to take as many repetitions of \w as possible without backtracking is: (?=(\w+))\1. Of course, we could take another pattern instead of \w.
That may seem odd, but it's actually a very simple transform.
Let's decipher it:

Lookahead ?= looks forward for the longest word \w+ starting at the current position.
The contents of parentheses with ?=... isn't memorized by the engine, so wrap \w+ into parentheses. Then the engine will memorize their contents
…And allow us to reference it in the pattern as \1.

That is: we look ahead – and if there's a word \w+, then match it as \1.
Why? That's because the lookahead finds a word \w+ as a whole and we capture it into the pattern with \1. So we essentially implemented a possessive plus + quantifier. It captures only the whole word \w+, not a part of it.
For instance, in the word JavaScript it may not only match Java, but leave out Script to match the rest of the pattern.
Here's the comparison of two patterns:

```
alert( "JavaScript".match(/\w+Script/)); // JavaScript
alert( "JavaScript".match(/(?=(\w+))\1Script/)); // null
```

In the first variant \w+ first captures the whole word JavaScript but then + backtracks character by character, to try to match the rest of the pattern, until it finally succeeds (when \w+ matches Java).
In the second variant (?=(\w+)) looks ahead and finds the word JavaScript, that is included into the pattern as a whole by \1, so there remains no way to find Script after it.

We can put a more complex regular expression into (?=(\w+))\1 instead of \w, when we need to forbid backtracking for + after it.

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.

Let's rewrite the first example using lookahead to prevent backtracking:

```
let regexp = /^((?=(\w+))\2\s?)*$/;

alert( regexp.test("A good string") ); // true

let str = "An input string that takes a long time or even makes this regex hang!";

alert( regexp.test(str) ); // false, works and fast!
```

Here \2 is used instead of \1, because there are additional outer parentheses. To avoid messing up with the numbers, we can give the parentheses a name, e.g. (?<word>\w+).

```
       // parentheses are named ?<word>, referenced as \k<word>
let regexp = /^((?=(?<word>\w+))\k<word>\s?)*$/;

let str = "An input string that takes a long time or even makes this regex hang!";

alert( regexp.test(str) ); // false

alert( regexp.test("A correct string") ); // true
```

The problem described in this article is called "catastrophic backtracking".
We covered two ways how to solve it:

Rewrite the regexp to lower the possible combinations count.
Prevent backtracking.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before
commenting…If you have suggestions what to improve - please submit a GitHub issue
or a pull request instead of commenting.If you can't understand something in the article
– please elaborate.To insert few words of code, use the <code> tag, for several lines –
wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin,
codepen…)var disqus_config = function() { if (!this.page) this.page = {};
Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-catastrophic-
backtracking","identifier":"\/regexp-catastrophic-backtracking"}); };var disqus_shortname
= "javascriptinfo";var disqus_enabled = true;
TutorialRegular expressions{"@context":"https://
schema.org","@type":"BreadcrumbList","itemListElement":
[{"@type":"ListItem","position":1,"name":"Tutorial","item":"https://javascript.info/"},
{"@type":"ListItem","position":2,"name":"Regular expressions","item":"https://
javascript.info/regular-expressions"}]}December 10, 2020Sticky flag "y", searching at
positionThe flag y allows to perform the search at the given position in the source string.
To grasp the use case of y flag, and better understand the ways of regexps, let's
explore a practical example.
One of common tasks for regexps is "lexical analysis": we get a text, e.g. in a
programming language, and need to find its structural elements. For instance, HTML
has tags and attributes, JavaScript code has functions, variables, and so on.
Writing lexical analyzers is a special area, with its own tools and algorithms, so we
don't go deep in there, but there's a common task: to read something at the given
position.
E.g. we have a code string let varName = "value", and we need to read the variable
name from it, that starts at position 4.
We'll look for variable name using regexp \w+. Actually, JavaScript variable names need
a bit more complex regexp for accurate matching, but here it doesn't matter.

A call to str.match(/\w+/) will find only the first word in the line (let). That's not it.
We can add the flag g. But then the call str.match(/\w+/g) will look for all words in the text, while we need one word at position 4. Again, not what we need.

So, how to search for a regexp exactly at the given position?
Let's try using method regexp.exec(str).
For a regexp without flags g and y, this method looks only for the first match, it works exactly like str.match(regexp).
…But if there's flag g, then it performs the search in str, starting from position stored in the regexp.lastIndex property. And, if it finds a match, then sets regexp.lastIndex to the index immediately after the match.
In other words, regexp.lastIndex serves as a starting point for the search, that each regexp.exec(str) call resets to the new value ("after the last match"). That's only if there's g flag, of course.
So, successive calls to regexp.exec(str) return matches one after another.
Here's an example of such calls:

```
        let str = 'let varName'; // Let's find all words in this string
let regexp = /\w+/g;

alert(regexp.lastIndex); // 0 (initially lastIndex=0)

let word1 = regexp.exec(str);
alert(word1[0]); // let (1st word)
alert(regexp.lastIndex); // 3 (position after the match)

let word2 = regexp.exec(str);
alert(word2[0]); // varName (2nd word)
alert(regexp.lastIndex); // 11 (position after the match)

let word3 = regexp.exec(str);
alert(word3); // null (no more matches)
alert(regexp.lastIndex); // 0 (resets at search end)
```

We can get all matches in the loop:

```
let str = 'let varName';
let regexp = /\w+/g;

let result;

while (result = regexp.exec(str)) {
  alert( `Found ${result[0]} at position ${result.index}` );
  // Found let at position 0, then
  // Found varName at position 4
}
```

Such use of regexp.exec is an alternative to method str.matchAll, with a bit more control over the process.

Let's go back to our task.

We can manually set lastIndex to 4, to start the search from the given position!

Like this:

```
let str = 'let varName = "value"';
```

```
let regexp = /\w+/g; // without flag "g", property lastIndex is ignored

regexp.lastIndex = 4;

let word = regexp.exec(str);
alert(word); // varName
```

Hooray! Problem solved!
We performed a search of \w+, starting from position regexp.lastIndex = 4.
The result is correct.
…But wait, not so fast.
Please note: the regexp.exec call starts searching at position lastIndex and then goes further. If there's no word at position lastIndex, but it's somewhere after it, then it will be found:

```
let str = 'let varName = "value"';

let regexp = /\w+/g;

// start the search from position 3
regexp.lastIndex = 3;

let word = regexp.exec(str);
// found the match at position 4
alert(word[0]); // varName
alert(word.index); // 4
```

For some tasks, including the lexical analysis, that's just wrong. We need to find a match exactly at the given position at the text, not somewhere after it. And that's what the flag y is for.
The flag y makes regexp.exec to search exactly at position lastIndex, not "starting from" it.

Here's the same search with flag y:

```
let str = 'let varName = "value"';

let regexp = /\w+/y;

regexp.lastIndex = 3;
alert( regexp.exec(str) ); // null (there's a space at position 3, not a word)

regexp.lastIndex = 4;
alert( regexp.exec(str) ); // varName (word at position 4)
```

As we can see, regexp /\w+/y doesn't match at position 3 (unlike the flag g), but matches at position 4.

Not only that's what we need, there's an important performance gain when using flag y. Imagine, we have a long text, and there are no matches in it, at all. Then a search with flag g will go till the end of the text and find nothing, and this will take significantly more time than the search with flag y, that checks only the exact position.

In tasks like lexical analysis, there are usually many searches at an exact position, to check what we have there. Using flag y is the key for correct implementations and a good performance.

Ctrl + !•Previous lessonCtrl + !'Next lessonShareTutorial mapCommentsread this before commenting…If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.If you can't understand something in the article – please elaborate.To insert few words of code, use the <code> tag, for several lines – wrap them in <pre> tag, for more than 10 lines – use a sandbox (plnkr, jsbin, codepen…)var disqus_config = function() { if (!this.page) this.page = {}; Object.assign(this.page, {"url":"https:\/\/javascript.info\/regexp-sticky","identifier":"\/regexp-sticky"}); };var disqus_shortname = "javascriptinfo";var disqus_enabled = true;