Part 3

Additional articles

JS

Ilya Kantor

Built at November 26, 2018

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

We constantly work to improve the tutorial. If you find any mistakes, please write at our github .

Animation

CSS and JavaScript animations.

Bezier curve

Bezier curves are used in computer graphics to draw shapes, for CSS animation and in many other places.

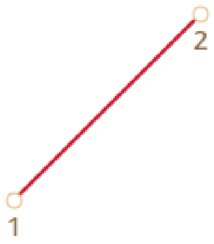
They are actually a very simple thing, worth to study once and then feel comfortable in the world of vector graphics and advanced animations.

Control points

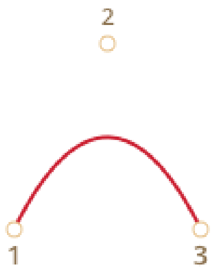
A bezier curve is defined by control points.

There may be 2, 3, 4 or more.

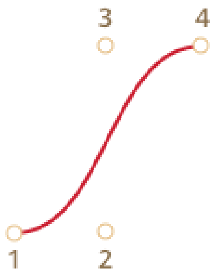
For instance, two points curve:



Three points curve:

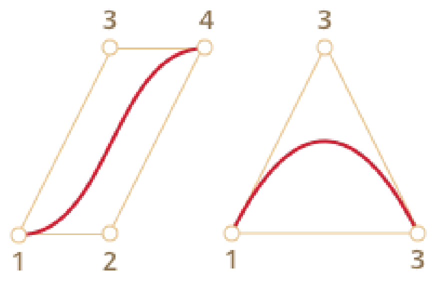


Four points curve:



If you look closely at these curves, you can immediately notice:

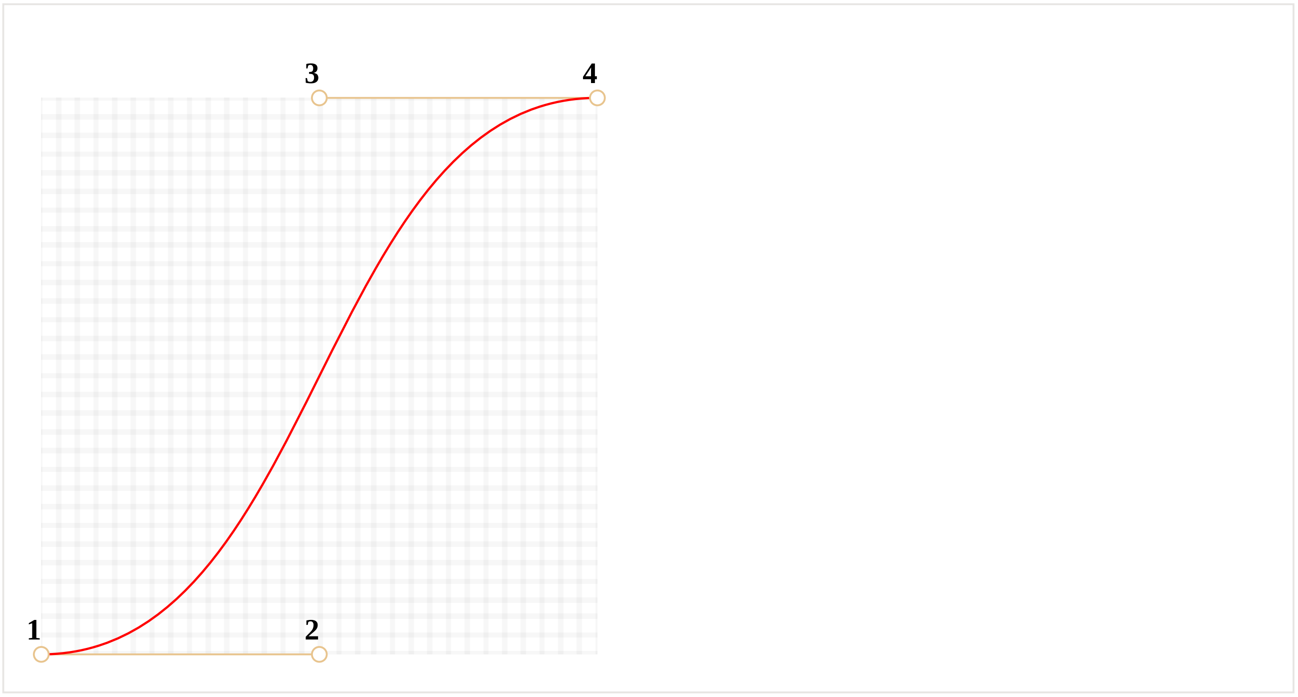
1. **Points are not always on curve.** That's perfectly normal, later we'll see how the curve is built.
2. **The curve order equals the number of points minus one**. For two points we have a linear curve (that's a straight line), for three points – quadratic curve (parabolic), for four points – cubic curve.
3. **A curve is always inside the convex hull of control points:**



Because of that last property, in computer graphics it's possible to optimize intersection tests. If convex hulls do not intersect, then curves do not either. So checking for the convex hulls intersection first can give a very fast "no intersection" result. Checking the intersection or convex hulls is much easier, because they are rectangles, triangles and so on (see the picture above), much simpler figures than the curve.

The main value of Bezier curves for drawing – by moving the points the curve is changing *in intuitively obvious way*.

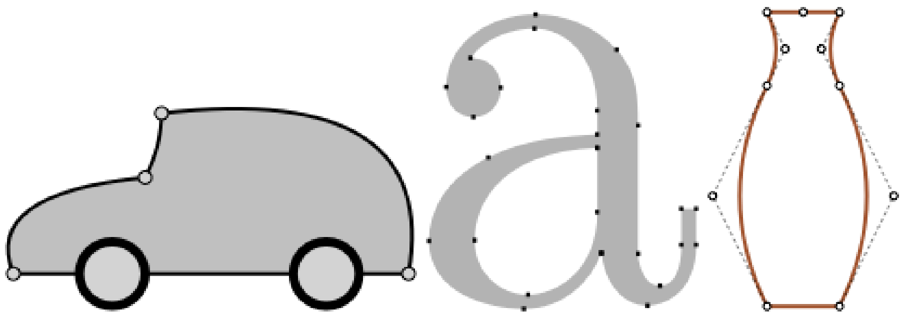
Try to move control points using a mouse in the example below:



As you can notice, the curve stretches along the tangential lines 1 → 2 and 3 → 4.

After some practice it becomes obvious how to place points to get the needed curve. And by connecting several curves we can get practically anything.

Here are some examples:



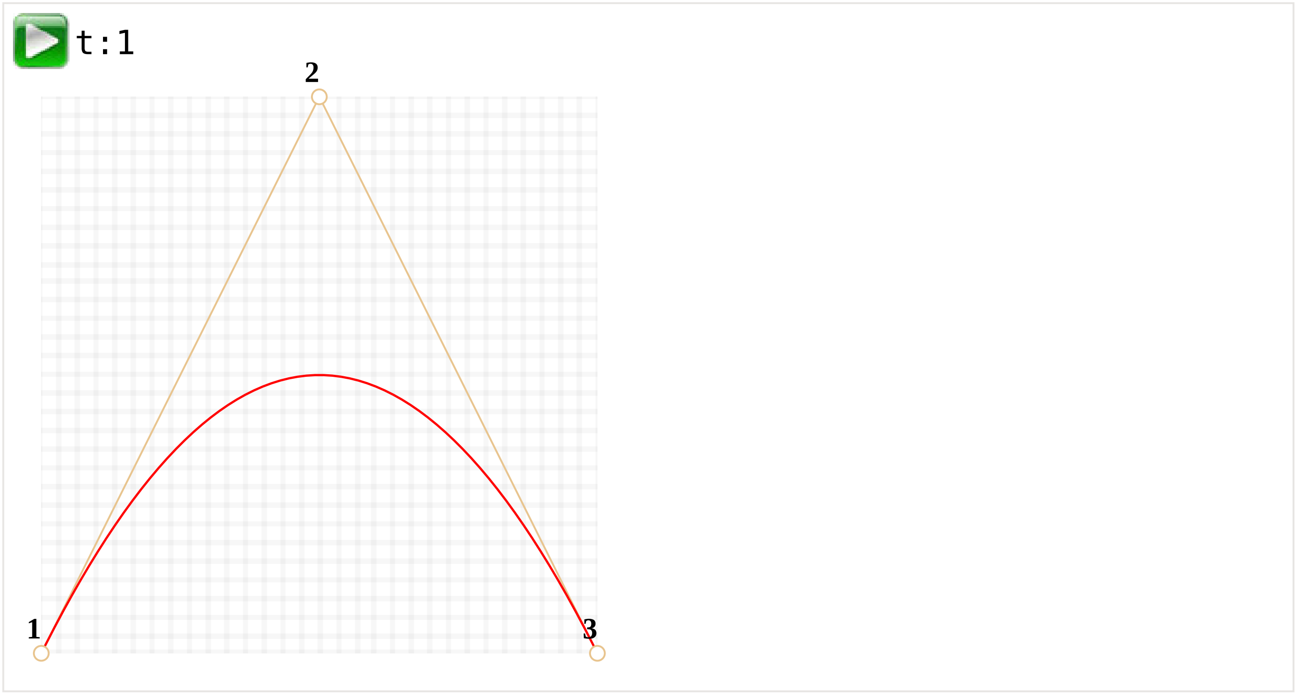
De Casteljau's algorithm

There's a mathematical formula for Bezier curves, but let's cover it a bit later, because De Casteljau's algorithm it is identical to the mathematical definition and visually shows how it is constructed.

First let's see the 3-points example.

Here's the demo, and the explanation follow.

Control points (1,2 and 3) can be moved by the mouse. Press the "play" button to run it.



De Casteljau's algorithm of building the 3-point bezier curve:

1. Draw control points. In the demo above they are labeled: `1`, `2`, `3`.
2. Build segments between control points 1 → 2 → 3. In the demo above they are brown.
3. The parameter `t` moves from `0` to `1`. In the example above the step `0.05` is used: the loop goes over `0, 0.05, 0.1, 0.15, ... 0.95, 1`.

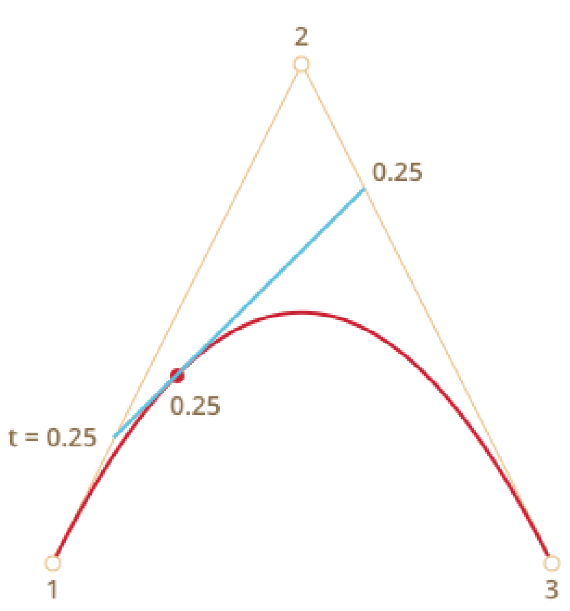
For each of these values of `t`:

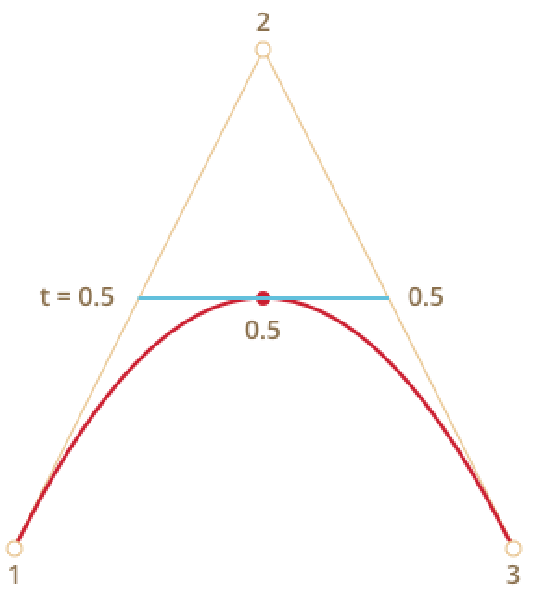
- On each brown segment we take a point located on the distance proportional to `t` from its beginning. As there are two segments, we have two points.

  For instance, for `t=0` – both points will be at the beginning of segments, and for `t=0.25` – on the 25% of segment length from the beginning, for `t=0.5` – 50% (the middle), for `t=1` – in the end of segments.

- Connect the points. On the picture below the connecting segment is painted blue.

For `t=0.25` **For `t=0.5`**

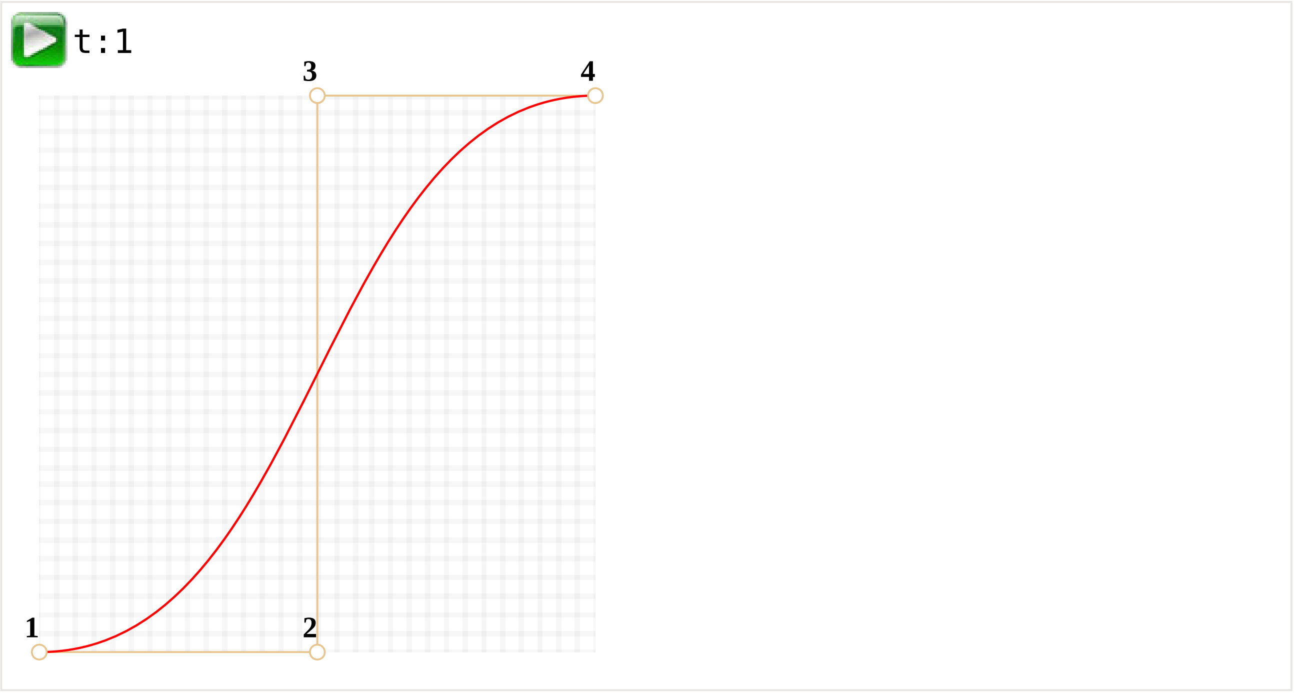




4. Now in the blue segment take a point on the distance proportional to the same value of `t`. That is, for `t=0.25` (the left picture) we have a point at the end of the left quarter of the segment, and for `t=0.5` (the right picture) – in the middle of the segment. On pictures above that point is red.
5. 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.

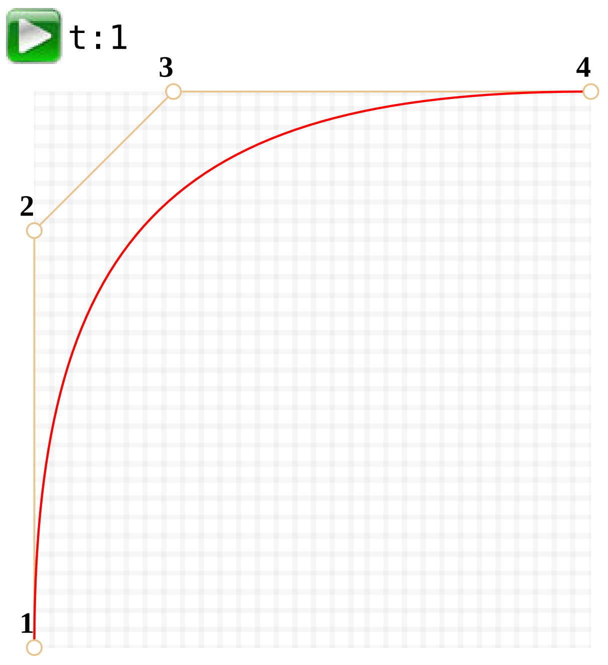
The algorithm is recursive and can be generalized for any number of control points.

Given N of control points:

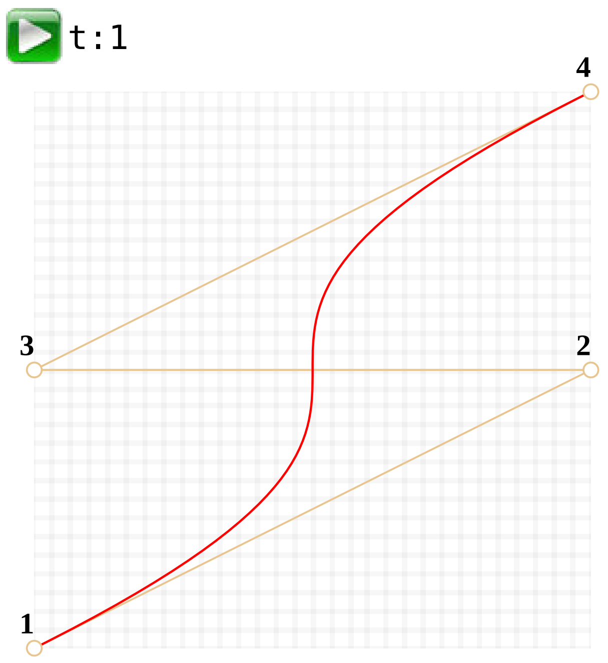
1. We connect them to get initially N-1 segments.
2. Then for each `t` from `0` to `1`, we take a point on each segment on the distance proportional to `t` and connect them. There will be N-2 segments.
3. Repeat step 2 until there is only one point.

These points make the curve.

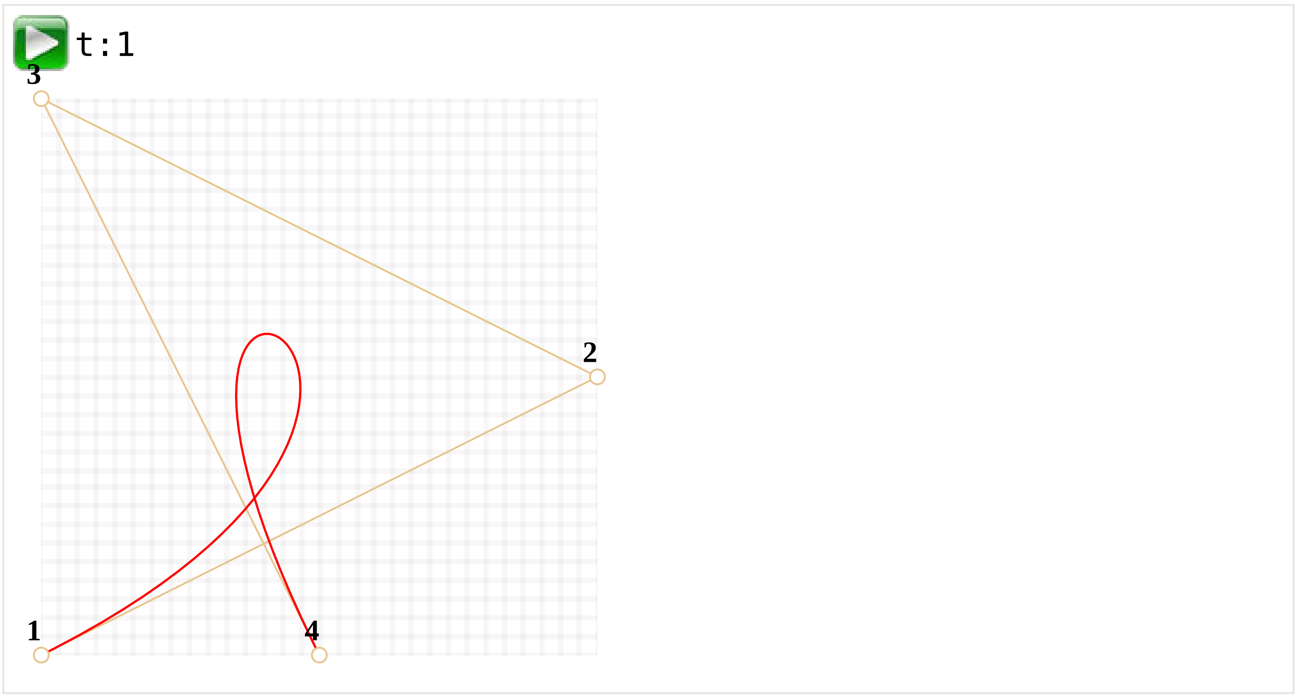
A curve that looks like `y=1/t`:



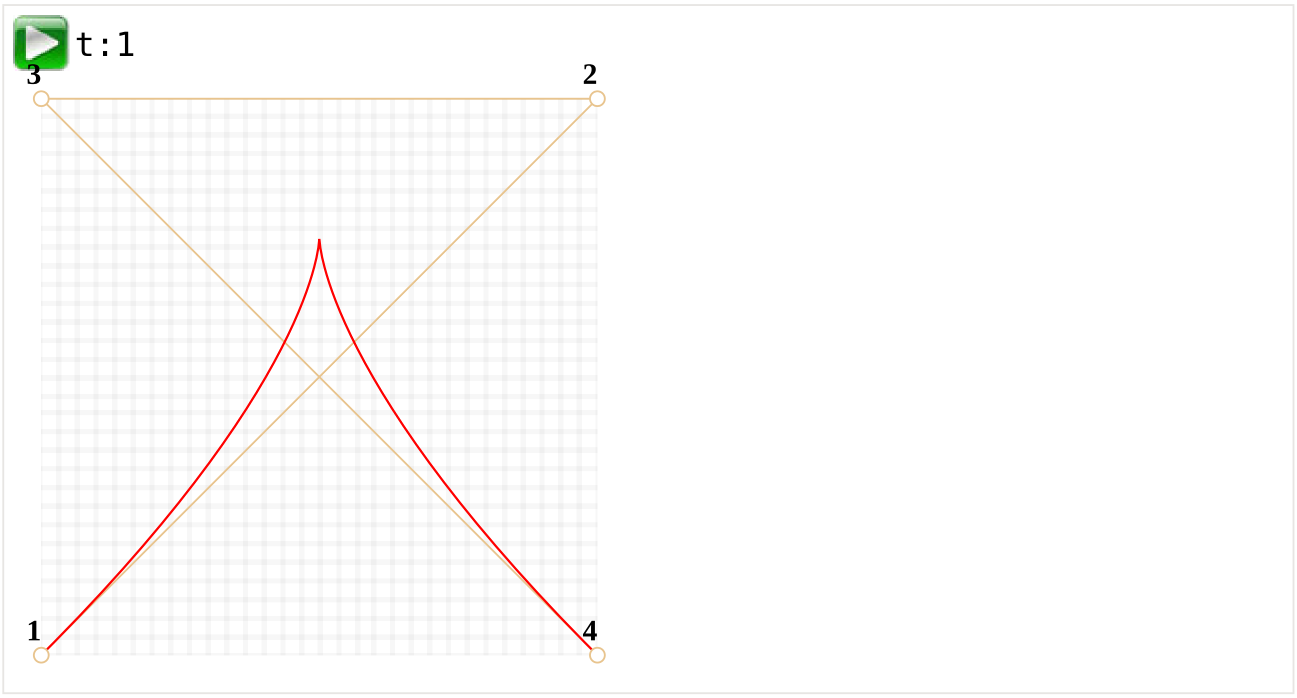
Zig-zag control points also work fine:



Making a loop is possible:



A non-smooth Bezier curve (yeah, that’s possible too):



As the algorithm is recursive, we can build Bezier curves of any order, that is: using 5, 6 or more control points. But in practice many points are less useful. Usually we take 2-3 points, and for complex lines glue several curves together. That’s simpler to develop and calculate.

How to draw a curve *through* given points?

We use control points for a Bezier curve. As we can see, they are not on the curve, except the first and the last ones.

Sometimes we have another task: to draw a curve *through several points*, so that all of them are on a single smooth curve. That task is called interpolation, and here we don't cover it.

There are mathematical formulas for such curves, for instance Lagrange polynomial. In computer graphics spline interpolation is often used to build smooth curves that connect many points.

Maths

A Bezier curve can be described using a mathematical formula.

As we saw – there's actually no need to know it, most people just draw the curve by moving points with a mouse. But if you're into maths – here it is.

Given the coordinates of control points P_i: the first control point has coordinates $P_1 = (x_1, y_1)$, the second: $P_2 = (x_2, y_2)$, and so on, the curve coordinates are described by the equation that depends on the parameter `t` from the segment `[0,1]`.

- The formula for a 2-points curve:

  $P = (1-t)P_1 + tP_2$

- For 3 control points:

  $P = (1-t)^2P_1 + 2(1-t)tP_2 + t^2P_3$

- For 4 control points:

  $P = (1-t)^3P_1 + 3(1-t)^2tP_2 +3(1-t)t^2P_3 + t^3P_4$

These are vector equations. In other words, we can put `x` and `y` instead of `P` to get corresponding coordinates.

For instance, the 3-point curve is formed by points `(x,y)` calculated as:

- $x = (1-t)^2x_1 + 2(1-t)tx_2 + t^2x_3$
- $y = (1-t)^2y_1 + 2(1-t)ty_2 + t^2y_3$

Instead of $x_1, y_1, x_2, y_2, x_3, y_3$ we should put coordinates of 3 control points, and then as `t` moves from `0` to `1`, for each value of `t` we'll have `(x,y)` of the curve.

For instance, if control points are `(0,0)`, `(0.5, 1)` and `(1, 0)`, the equations become:

- $x = (1-t)^2 * 0 + 2(1-t)t * 0.5 + t^2 * 1 = (1-t)t + t^2 = t$
- $y = (1-t)^2 * 0 + 2(1-t)t * 1 + t^2 * 0 = 2(1-t)t = -t^2 + 2t$

Now as `t` runs from `0` to `1`, the set of values `(x,y)` for each `t` forms the curve for such control points.

Summary

Bezier curves are defined by their control points.

We saw two definitions of Bezier curves:

1. Using a mathematical formulas.
2. Using a drawing process: De Casteljau’s algorithm

Good properties of Bezier curves:

- We can draw smooth lines with a mouse by moving around control points.
- Complex shapes can be made of several Bezier curves.

Usage:

- In computer graphics, modeling, vector graphic editors. Fonts are described by Bezier curves.
- In web development – for graphics on Canvas and in the SVG format. By the way, “live” examples above are written in SVG. They are actually a single SVG document that is given different points as parameters. You can open it in a separate window and see the source: demo.svg.
- In CSS animation to describe the path and speed of animation.

CSS-animations

CSS animations allow to do simple animations without JavaScript at all.

JavaScript can be used to control CSS animation and make it even better with a little of code.

CSS transitions

The idea of CSS transitions is simple. We describe a property and how its changes should be animated. When the property changes, the browser paints the animation.

That is: all we need is to change the property. And the fluent transition is made by the browser.

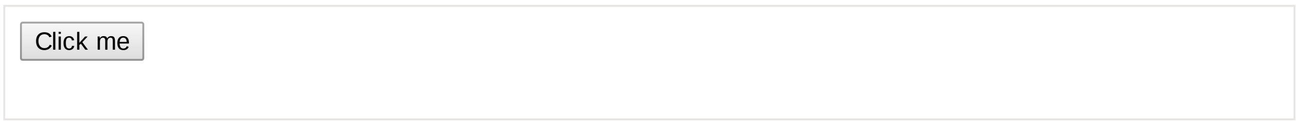
For instance, the CSS below animates changes of `background-color` for 3 seconds:

```css
.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:

```html
<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>
```



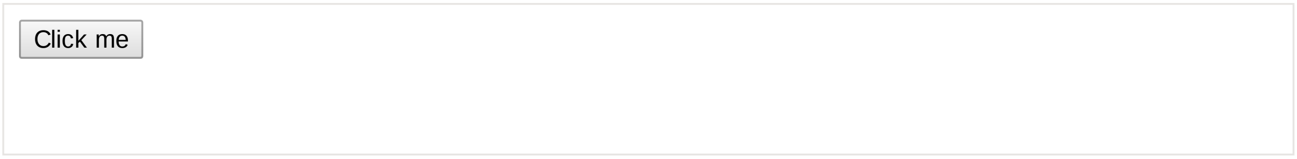
There are 4 properties to describe CSS transitions:

- `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 to declare them together in the order: `property duration timing-function delay`, and also animate multiple properties at once.

For instance, this button animates both `color` and `font-size`:

```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-property

In `transition-property` we write a list of property to animate, for instance: `left`, `margin-left`, `height`, `color`.

Not all properties can be animated, but many of them. The value `all` means "animate all properties".

transition-duration

In `transition-duration` we can specify how long the animation should take. The time should be in CSS time format: in seconds `s` or milliseconds `ms`.

transition-delay

In `transition-delay` we can specify the delay *before* the animation. For instance, if `transition-delay: 1s`, then animation starts after 1 second after the change.

Negative values are also possible. Then the animation starts from the middle. For instance, if `transition-duration` is `2s`, and the delay is `-1s`, then the animation takes 1 second and starts from the half.

Here's the animation shifts numbers from `0` to `9` using CSS `translate` property:

http://plnkr.co/edit/tRHA6fkSPUe9cjk35zPL?p=preview

The `transform` property is animated like this:

```css
#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:

```js
stripe.classList.add('animate');
```

We can also start it “from the middle”, from the exact number, e.g. corresponding to the current second, using the negative `transition-delay`.

Here if you click the digit – it starts the animation from the current second:

http://plnkr.co/edit/zpqja4CejOmTApUXPxwE?p=preview

JavaScript does it by an extra line:

```js
stripe.onclick = function() {
  let sec = new Date().getSeconds() % 10;
  // for instance, -3s here starts the animation from the 3rd second
  stripe.style.transitionDelay = '-' + sec + 's';
  stripe.classList.add('animate');
};
```

transition-timing-function

Timing function describes how the animation process is distributed along the time. Will it start slowly and then go fast or vise versa.

That’s the most complicated property from the first sight. But it becomes very simple if we devote a bit time to it.

That property accepts two kinds of values: a Bezier curve or steps. Let’s start from the curve, as it’s used more often.

Bezier curve

The timing function can be set as a Bezier curve with 4 control points that satisfies the conditions:

1. First control point: `(0,0)`.
2. Last control point: `(1,1)`.
3. For intermediate points values of `x` must be in the interval `0..1`, `y` can be anything.

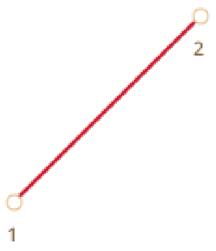
The syntax for a Bezier curve in CSS: `cubic-bezier(x2, y2, x3, y3)`. Here we need to specify only 2nd and 3rd control points, because the 1st one is fixed to `(0,0)` and the 4th one is `(1,1)`.

The timing function describes how fast the animation process goes in time.

- The `x` axis is the time: `0` – the starting moment, `1` – the last moment of `transition-duration`.
- The `y` axis specifies the completion of the process: `0` – the starting value of the property, `1` – the final value.

The simplest variant is when the animation goes uniformly, with the same linear speed. That can be specified by the curve `cubic-bezier(0, 0, 1, 1)`.

Here's how that curve looks:



…As we can see, it's just a straight line. As the time (`x`) passes, the completion (`y`) of the animation steadily goes from `0` to `1`.

The train in the example below goes from left to right with the permanent speed (click it):

http://plnkr.co/edit/BKXxsW1mgxIZvhcpUcj4?p=preview

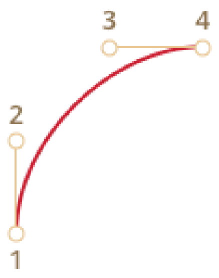
The CSS `transition` is based on that curve:

```
.train {
  left: 0;
  transition: left 5s cubic-bezier(0, 0, 1, 1);
  /* JavaScript sets left to 450px */
}
```

…And how can we show a train slowing down?

We can use another Bezier curve: `cubic-bezier(0.0, 0.5, 0.5 ,1.0)`.

The graph:



As we can see, the process starts fast: the curve soars up high, and then slower and slower.

Here’s the timing function in action (click the train):

http://plnkr.co/edit/EBBtkTnI1l5096SHLcq8?p=preview

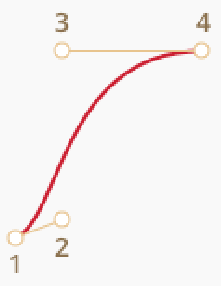
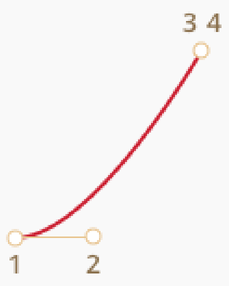
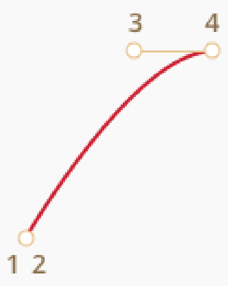
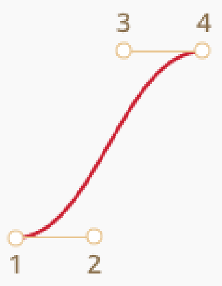
CSS:

```
.train {
  left: 0;
  transition: left 5s cubic-bezier(0, .5, .5, 1);
  /* JavaScript sets left to 450px */
}
```

There are several built-in curves: `linear`, `ease`, `ease-in`, `ease-out` and `ease-in-out`.

The `linear` is a shorthand for `cubic-bezier(0, 0, 1, 1)` – a straight line, we saw it already.

Other names are shorthands for the following `cubic-bezier`:

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

But it looks a bit differently.

A Bezier curve can make the animation “jump out” of its range.

The control points on the curve can have any `y` coordinates: even negative or huge. Then the Bezier curve would also jump very low or high, making the animation go beyond its normal range.

In the example below the animation code is:

```
.train {
  left: 100px;
  transition: left 5s cubic-bezier(.5, -1, .5, 2);
  /* JavaScript sets left to 400px */
}
```

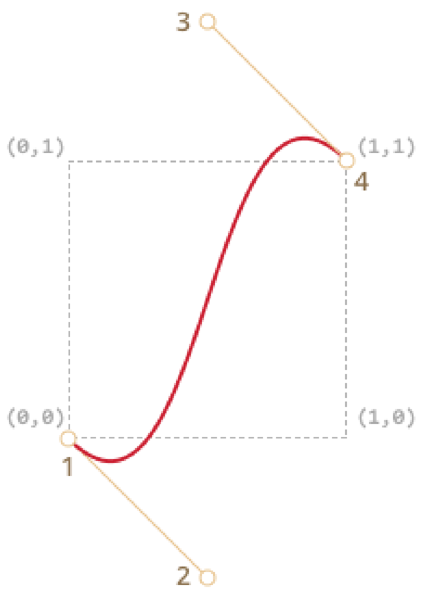
The property `left` should animate from `100px` to `400px`.

But if you click the train, you’ll see that:

- First, the train goes *back*: `left` becomes less than `100px`.
- Then it goes forward, a little bit farther than `400px`.
- And then back again – to `400px`.

http://plnkr.co/edit/TjXgcacdDDsFyYHb4Lnl?p=preview

Why it happens – pretty obvious if we look at the graph of the given Bezier curve:



We moved the `y` coordinate of the 2nd point below zero, and for the 3rd point we made put it over `1`, so the curve goes out of the “regular” quadrant. The `y` is out of the “standard” range `0..1`.

As we know, `y` measures “the completion of the animation process”. The value `y = 0` corresponds to the starting property value and `y = 1` – the ending value. So values `y<0` move the property lower than the starting `left` and `y>1` – over the final `left`.

That’s a “soft” variant for sure. If we put `y` values like `-99` and `99` then the train would jump out of the range much more.

But how to make the Bezier curve for a specific task? There are many tools. For instance, we can do it on the site http://cubic-bezier.com/ .

Steps

Timing function `steps(number of steps[, start/end])` allows to split animation into steps.

Let’s see that in an example with digits.

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

http://plnkr.co/edit/iyY2pj0vD8CcbFCuqnsI?p=preview

We’ll make the digits appear in a discrete way by making the part of the list outside of the red “window” invisible and shifting the list to the left with each step.

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

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

In action:

http://plnkr.co/edit/6VBxPjYIojjUL5vX8UvS?p=preview

The first argument of `steps(9, start)` is the number of steps. The transform will be split into 9 parts (10% each). The time interval is automatically divided into 9 parts as well, so `transition: 9s` gives us 9 seconds for the whole animation – 1 second per digit.

The second argument is one of two words: `start` or `end`.

The `start` means that in the beginning of animation we need to do make the first step immediately.

We can observe that during the animation: when we click on the digit it changes to `1` (the first step) immediately, and then changes in the beginning of the next second.

The process is progressing like this:

- `0s` – `-10%` (first change in the beginning of the 1st second, immediately)
- `1s` – `-20%`
- …
- `8s` – `-80%`
- (the last second shows the final value).

The alternative value `end` would mean that the change should be applied not in the beginning, but at the end of each second.

So the process would go like this:

- `0s` – `0`
- `1s` – `-10%` (first change at the end of the 1st second)
- `2s` – `-20%`
- …
- `9s` – `-90%`

Here's `step(9, end)` in action (note the pause between the first digit change):

http://plnkr.co/edit/I3SoddMNBYDKxH2HeFak?p=preview

There are also shorthand values:

- `step-start` – is the same as `steps(1, start)`. That is, the animation starts immediately and takes 1 step. So it starts and finishes immediately, as if there were no animation.
- `step-end` – the same as `steps(1, end)`: make the animation in a single step at the end of `transition-duration`.

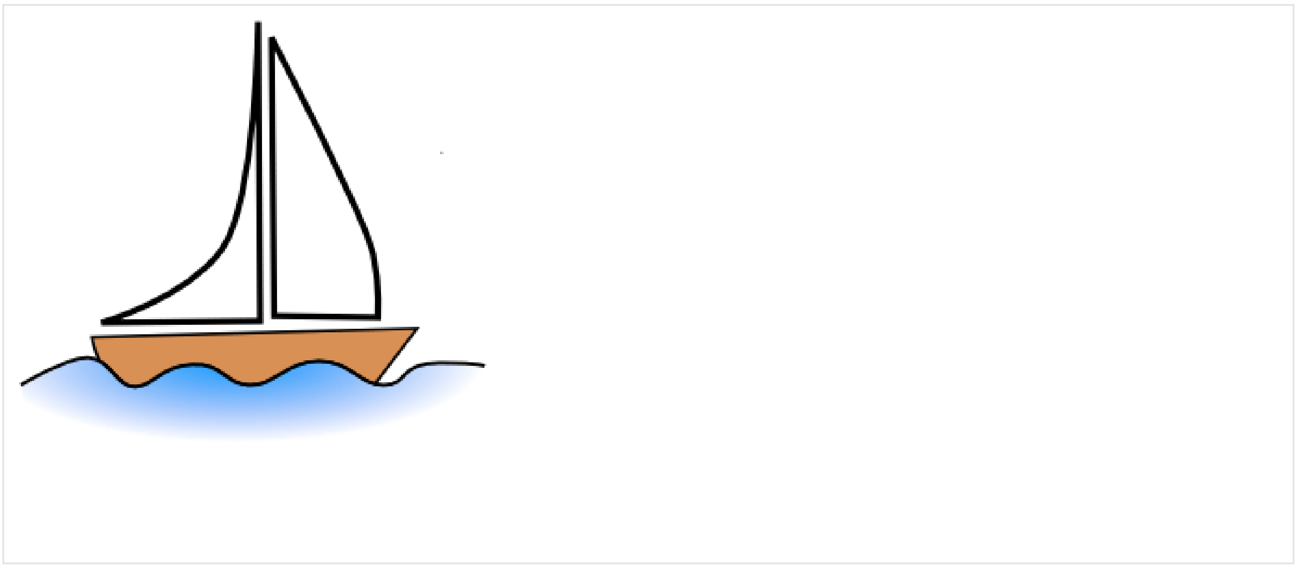
These values are rarely used, because that's not really animation, but rather a single-step change.

Event transitionend

When the CSS animation finishes the `transitionend` event triggers.

It is widely used to do an action after the animation is done. Also we can join animations.

For instance, the ship in the example below starts to swim there and back on click, each time farther and farther to the right:



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

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

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

  }

  go();

  boat.addEventListener('transitionend', function() {
    times++;
    go();
  });
};
```

The event object for `transitionend` has few specific properties:

`event.propertyName`

The property that has finished animating. Can be good if we animate multiple properties simultaneously.

`event.elapsedTime`

The time (in seconds) that the animation took, without `transition-delay`.

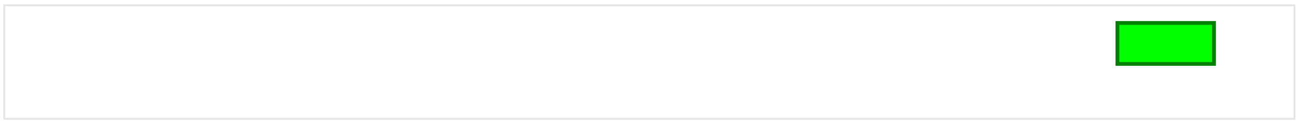
Keyframes

We can join multiple simple animations together using the `@keyframes` CSS rule.

It specifies the “name” of the animation and rules: what, when and where to animate. Then using the `animation` property we attach the animation to the element and specify additional parameters for it.

Here’s an example with explanations:

```
<div class="progress"></div>

<style>
  @keyframes go-left-right {        /* give it a name: "go-left-right" */
    from { left: 0px; }             /* animate from left: 0px */
    to { left: calc(100% - 50px); } /* animate to left: 100%-50px */
  }

  .progress {
    animation: go-left-right 3s infinite alternate;
    /* apply the animation "go-left-right" to the element
       duration 3 seconds
       number of times: infinite
       alternate direction every time
    */

    position: relative;
    border: 2px solid green;
    width: 50px;
    height: 20px;
    background: lime;
  }
</style>
```



There are many articles about `@keyframes` and a detailed specification.

Probably you won’t need `@keyframes` often, unless everything is in the constant move on your sites.

Summary

CSS animations allow to smoothly (or not) animate changes of one or multiple CSS properties.

They are good for most animation tasks. We’re also able to use JavaScript for animations, the next chapter is devoted to that.

Limitations of CSS animations compared to JavaScript animations:

Merits

- Simple things done simply.
- Fast and lightweight for CPU.

Demerits

- JavaScript animations are flexible. They can implement any animation logic, like an “explosion” of an element.
- Not just property changes. We can create new elements in JavaScript for purposes of animation.

The majority of animations can be implemented using CSS as described in this chapter. And `transitionend` event allows to run JavaScript after the animation, so it integrates fine with the code.

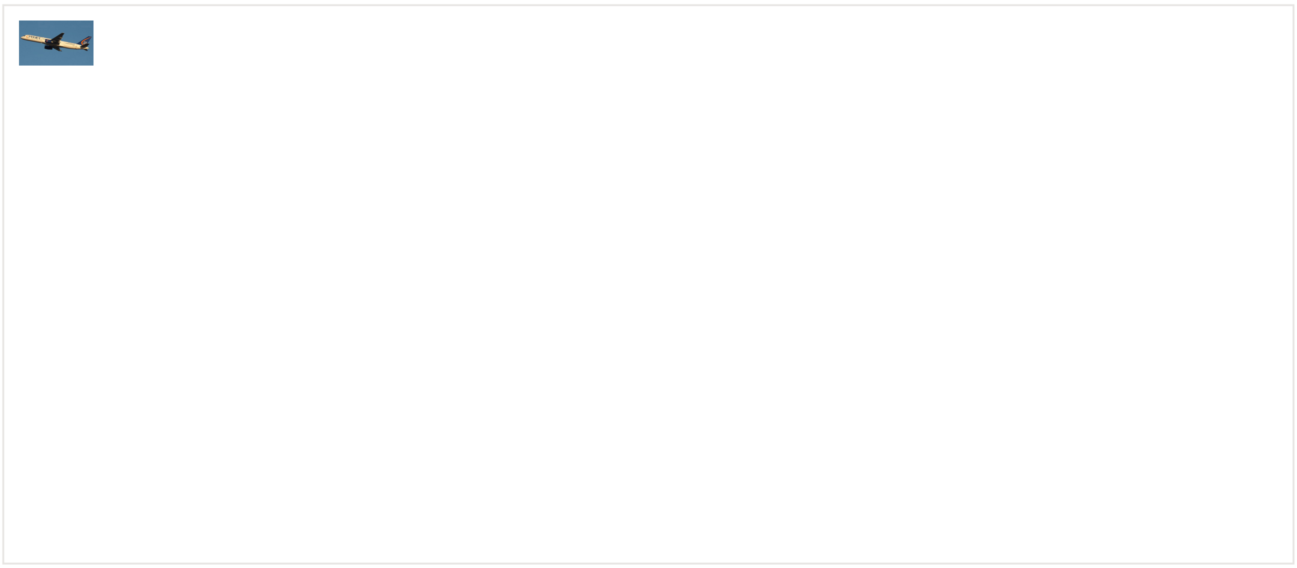
But in the next chapter we’ll do some JavaScript animations to cover more complex cases.

Tasks

Animate a plane (CSS)

importance: 5

Show the animation like on the picture below (click the plane):



- The picture grows on click from `40x24px` to `400x240px` (10 times larger).
- The animation takes 3 seconds.
- At the end output: “Done!”.
- During the animation process, there may be more clicks on the plane. They shouldn’t “break” anything.

Open a sandbox for the task.

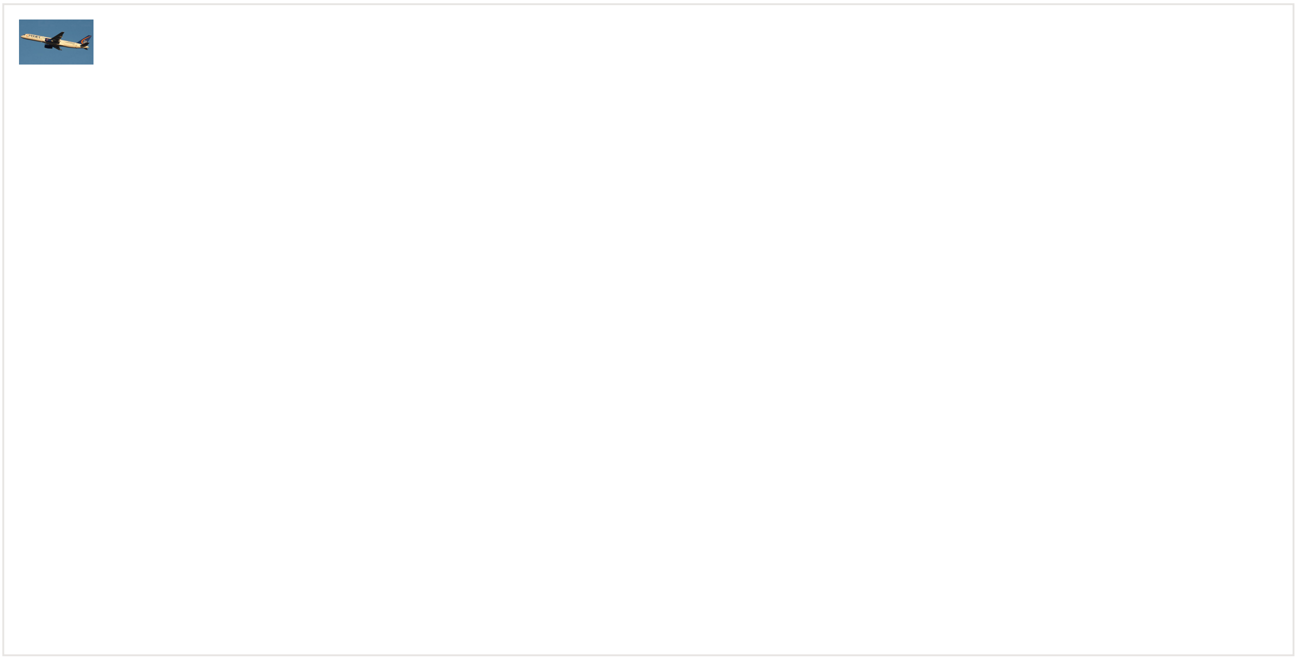
To solution

Animate the flying plane (CSS)

importance: 5

Modify the solution of the previous task Animate a plane (CSS) to make the plane grow more than it’s original size 400x240px (jump out), and then return to that size.

Here’s how it should look (click on the plane):



Take the solution of the previous task as the source.

To solution

Animated circle

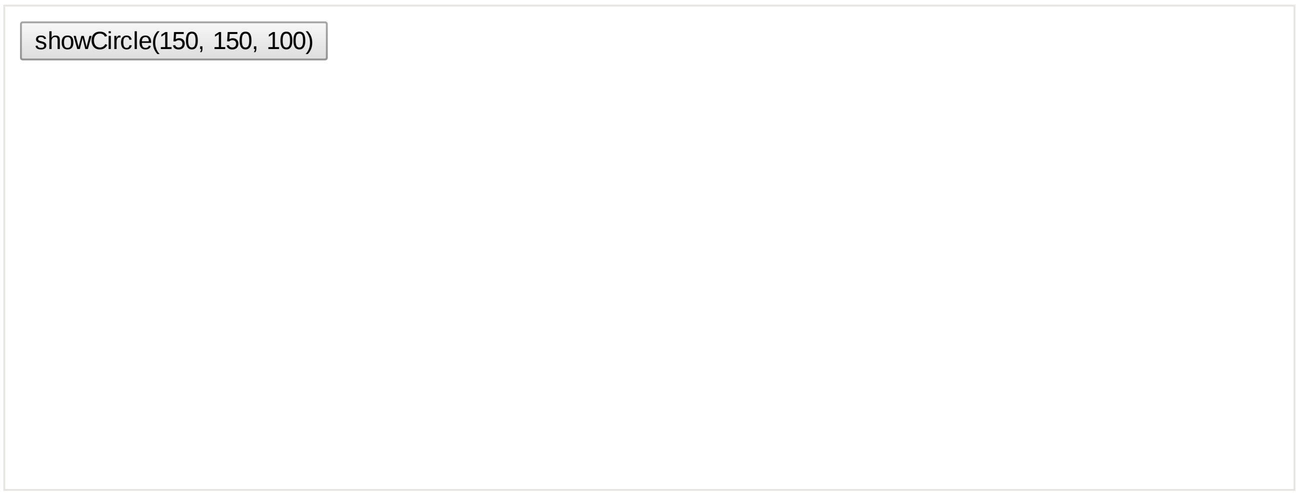
importance: 5

Create a function `showCircle(cx, cy, radius)` that shows an animated growing circle.

- `cx,cy` are window-relative coordinates of the center of the circle,

- `radius` is the radius of the circle.

Click the button below to see how it should look like:



The source document has an example of a circle with right styles, so the task is precisely to do the animation right.

Open a sandbox for the task.

To solution

JavaScript animations

JavaScript animations can handle things that CSS can't.

For instance, moving along a complex path, with a timing function different from Bezier curves, or an animation on a canvas.

Using setInterval

An animation can be implemented as a sequence of frames – usually small changes to HTML/CSS properties.

For instance, changing `style.left` from `0px` to `100px` moves the element. And if we increase it in `setInterval`, changing by `2px` with a tiny delay, like 50 times per second, then it looks smooth. That's the same principle as in the cinema: 24 or more 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:

http://plnkr.co/edit/rah4Qvue9bwrmtS3ASKt?p=preview

Using requestAnimationFrame

Let's imagine we have several animations running simultaneously.

If we run them separately, then even though each one has `setInterval(..., 20)`, then the browser would have to repaint much more often than every `20ms`.

That's because they have different starting time, so “every 20ms” differs between different animations. The intervals are not alignned. 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 smoother for people).

There’s one more thing to keep in mind. Sometimes when CPU is overloaded, or there are other reasons to redraw less often (like when the browser tab is hidden), so we really shouldn’t run it every `20ms`.

But how do we know about that in JavaScript? There’s a specification Animation timing that provides the function `requestAnimationFrame`. It addresses all these issues and even more.

The syntax:

```
let requestId = requestAnimationFrame(callback)
```

That schedules the `callback` function to run in the closest time when the browser wants to do animation.

If we do changes in elements in `callback` then they will be grouped together with other `requestAnimationFrame` callbacks and with CSS animations. So there will be one geometry recalculation and repaint instead of many.

The returned value `requestId` can be used to cancel the call:

```
// cancel the scheduled execution of callback
cancelAnimationFrame(requestId);
```

The `callback` gets one argument – the time passed from the beginning of the page load in microseconds. This time can also be obtained by calling performance.now().

Usually `callback` runs very soon, unless the CPU is overloaded or the laptop battery is almost discharged, or there’s another reason.

The code below shows the time between first 10 runs for `requestAnimationFrame`. Usually it’s 10-20ms:

```
<script>
  let prev = performance.now();
  let times = 0;

  requestAnimationFrame(function measure(time) {
    document.body.insertAdjacentHTML("beforeEnd", Math.floor(time - prev) + " ");
    prev = time;

    if (times++ < 10) requestAnimationFrame(measure);
  })
</script>
```

Structured animation

Now we can make a more universal animation function based on `requestAnimationFrame`:

```
function animate({timing, draw, duration}) {

  let start = performance.now();

  requestAnimationFrame(function animate(time) {
    // timeFraction goes from 0 to 1
    let timeFraction = (time - start) / duration;
    if (timeFraction > 1) timeFraction = 1;

    // calculate the current animation state
    let progress = timing(timeFraction)

    draw(progress); // draw it

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

  });
}
```

Function `animate` accepts 3 parameters that essentially describes the animation:

`duration`

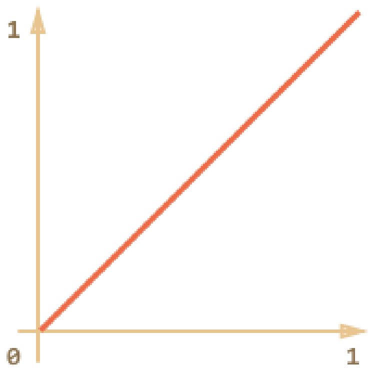
Total time of animation. Like, `1000`.

`timing(timeFraction)`

Timing function, like CSS-property `transition-timing-function` that gets the fraction of time that passed (`0` at start, `1` at the end) and returns the animation completion (like `y` on the Bezier curve).

For instance, a linear function means that the animation goes on uniformly with the same speed:

```
function linear(timeFraction) {
  return timeFraction;
}
```

It’s graph: 

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:

http://plnkr.co/edit/5l241DCQmzPNofVr2wfk?p=preview

The code for it:

```
animate({
  duration: 1000,
  timing(timeFraction) {
    return timeFraction;
  },
  draw(progress) {
    elem.style.width = progress * 100 + '%';
  }
});
```

Unlike CSS animation, we can make any timing function and any drawing function here. The timing function is not limited by Bezier curves. And `draw` can go beyond properties, create new elements for like fireworks animation or something.

Timing functions

We saw the simplest, linear timing function above.

Let's see more of them. We'll try movement animations with different timing functions to see how they work.

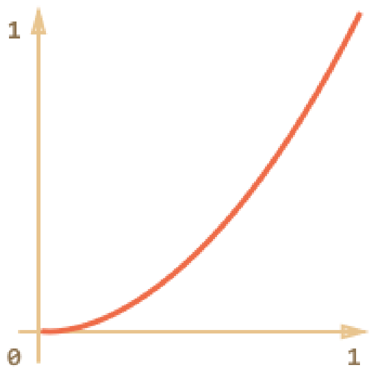
Power of n

If we want to speed up the animation, we can use `progress` in the power `n`.

For instance, a parabolic curve:

```
function quad(timeFraction) {
  return Math.pow(timeFraction, 2)
}
```

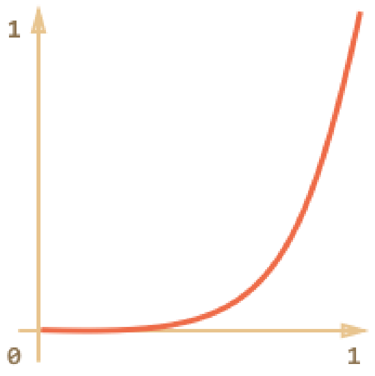
The graph:



See in action (click to activate):

…Or the cubic curve or event greater `n`. Increasing the power makes it speed up faster.

Here's the graph for `progress` in the power `5`:



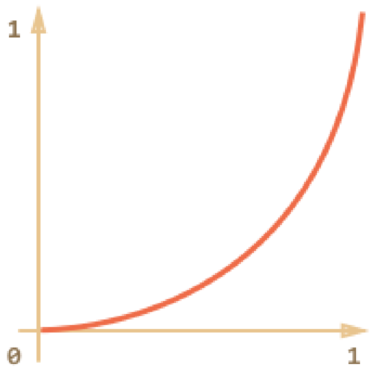
In action:

The arc

Function:

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

The graph:



Back: bow shooting

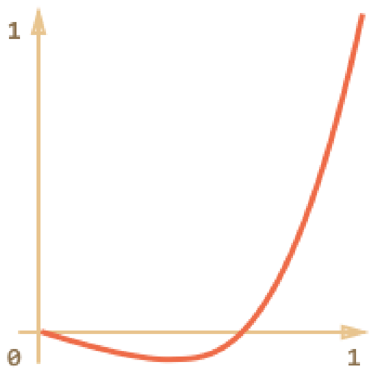
This function does the “bow shooting”. First we “pull the bowstring”, and then “shoot”.

Unlike previous functions, it depends on an additional parameter `x`, the “elasticity coefficient”. The distance of “bowstring pulling” is defined by it.

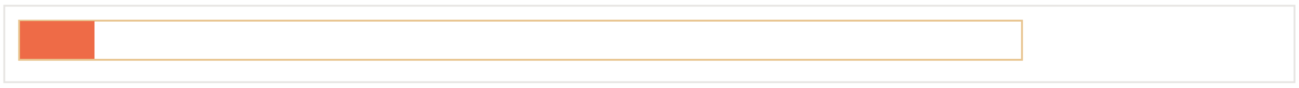
The code:

```
function back(x, timeFraction) {
  return Math.pow(timeFraction, 2) * ((x + 1) * timeFraction - x)
}
```

The graph for `x = 1.5`:



For animation we use it with a specific value of `x`. Example for `x = 1.5`:



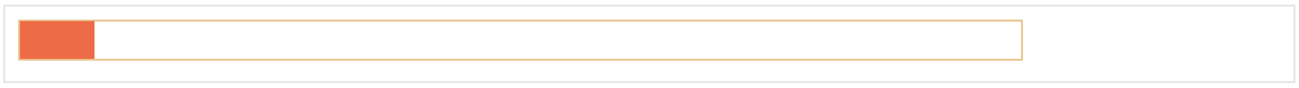
Bounce

Imagine we are dropping a ball. It falls down, then bounces back a few times and stops.

The `bounce` function does the same, but in the reverse order: “bouncing” starts immediately. It uses few special coefficients for that:

```
function bounce(timeFraction) {
  for (let a = 0, b = 1, result; 1; a += b, b /= 2) {
    if (timeFraction >= (7 - 4 * a) / 11) {
      return -Math.pow((11 - 6 * a - 11 * timeFraction) / 4, 2) + Math.pow(b, 2)
    }
  }
}
```

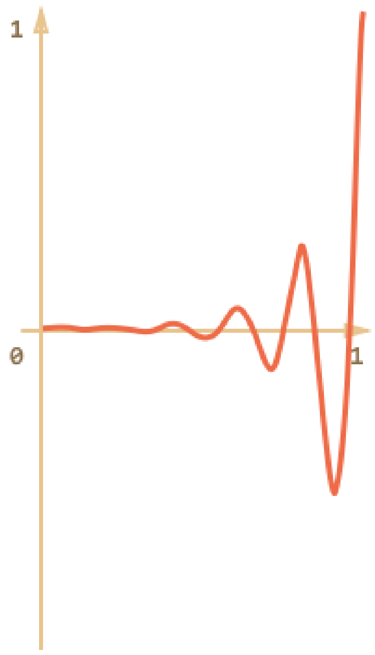
In action:



Elastic animation

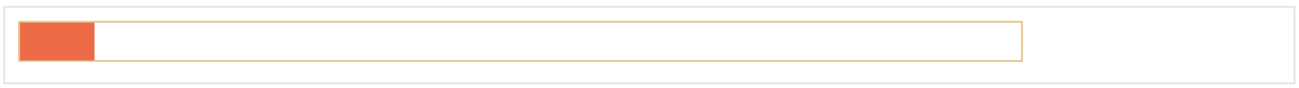
One more “elastic” function that accepts an additional parameter `x` for the “initial range”.

```
function elastic(x, timeFraction) {
  return Math.pow(2, 10 * (timeFraction - 1)) * Math.cos(20 * Math.PI * x / 3 * timeFr
}
```



The graph for `x=1.5`:

In action for `x=1.5`:



Reversal: ease*

So we have a collection of timing functions. Their direct application is called “easeIn”.

Sometimes we need to show the animation in the reverse order. That’s done with the “easeOut” transform.

easeOut

In the “easeOut” mode the `timing` function is put into a wrapper `timingEaseOut`:

```
timingEaseOut(timeFraction) = 1 - timing(1 - timeFraction)
```

In other words, we have a “transform” function `makeEaseOut` that takes a “regular” timing function and returns the wrapper around it:

```
// accepts a timing function, returns the transformed variant
function makeEaseOut(timing) {
  return function(timeFraction) {
    return 1 - timing(1 - timeFraction);
  }
}
```

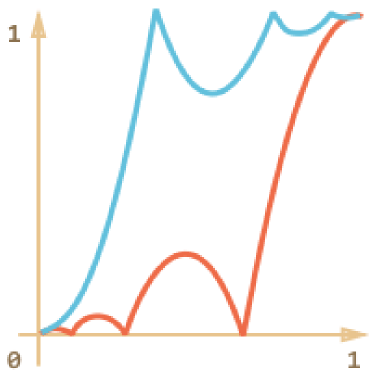
For instance, we can take the `bounce` function described above and apply it:

```
let bounceEaseOut = makeEaseOut(bounce);
```

Then the bounce will be not in the beginning, but at the end of the animation. Looks even better:

http://plnkr.co/edit/opuSRjafyQ8Y41QXOolD?p=preview

Here we can see how the transform changes the behavior of the function:



If there's an animation effect in the beginning, like bouncing – it will be shown at the end.

In the graph above the regular bounce has the red color, and the easeOut bounce is blue.

- Regular bounce – the object bounces at the bottom, then at the end sharply jumps to the top.
- After `easeOut` – it first jumps to the top, then bounces there.

easeInOut

We also can show the effect both in the beginning and the end of the animation. The transform is called "easeInOut".

Given the timing function, we calculate the animation state like this:

```
if (timeFraction <= 0.5) { // first half of the animation
  return timing(2 * timeFraction) / 2;
} else { // second half of the animation
  return (2 - timing(2 * (1 - timeFraction))) / 2;
}
```

The wrapper code:

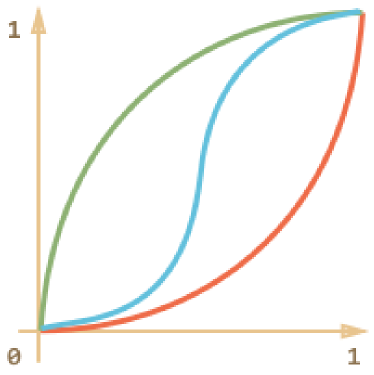
```
function makeEaseInOut(timing) {
  return function(timeFraction) {
    if (timeFraction < .5)
      return timing(2 * timeFraction) / 2;
    else
      return (2 - timing(2 * (1 - timeFraction))) / 2;
  }
}

bounceEaseInOut = makeEaseInOut(bounce);
```

In action, `bounceEaseInOut`:

http://plnkr.co/edit/F7NuLTRQblC8EgZr8ltV?p=preview

The “easeInOut” transform joins two graphs into one: `easeIn` (regular) for the first half of the animation and `easeOut` (reversed) – for the second part.

The effect is clearly seen if we compare the graphs of `easeIn`, `easeOut` and `easeInOut` of the `circ` timing function:



- Red is the regular variantof `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 write the proper `draw`.

Here’s the animated “bouncing” text typing:

http://plnkr.co/edit/IX995Jbip9id4R2Vwer9?p=preview

Summary

For animations that CSS can’t handle well, or those that need tight control, JavaScript can help. JavaScript animations should be implemented via `requestAnimationFrame`. That built-in method allows to setup a callback function to run when the browser will be preparing a repaint. Usually that's very soon, but the exact time depends on the browser.

When a page is in the background, there are no repaints at all, so the callback won't run: the animation will be suspended and won't consume resources. That's great.

Here's the helper `animate` function to setup most animations:

```
function animate({timing, draw, duration}) {

  let start = performance.now();

  requestAnimationFrame(function animate(time) {
    // timeFraction goes from 0 to 1
    let timeFraction = (time - start) / duration;
    if (timeFraction > 1) timeFraction = 1;

    // calculate the current animation state
    let progress = timing(timeFraction);

    draw(progress); // draw it

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

  });
}
```

Options:

- `duration` – the total animation time in ms.
- `timing` – the function to calculate animation progress. Gets a time fraction from 0 to 1, returns the animation progress, usually from 0 to 1.
- `draw` – the function to draw the animation.

Surely we could improve it, add more bells and whistles, but JavaScript animations are not applied on a daily basis. They are used to do something interesting and non-standard. So you'd want to add the features that you need when you need them.

JavaScript animations can use any timing function. We covered a lot of examples and transformations to make them even more versatile. Unlike CSS, we are not limited to Bezier curves here.

The same is about `draw`: we can animate anything, not just CSS properties.

✓ Tasks

Animate the bouncing ball

importance: 5

Make a bouncing ball. Click to see how it should look:

Open a sandbox for the task.

To solution

Animate the ball bouncing to the right

importance: 5

Make the ball bounce to the right. Like this:

Write the animation code. The distance to the left is `100px`.

Take the solution of the previous task Animate the bouncing ball as the source.

To solution

Frames and windows

Popups and window methods

A popup window is one of the oldest methods to show additional document to user.

Basically, you just run:

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

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

Popup blocking

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: JavaScript is able to send requests for server, so popups are rarely used. But sometimes they are still handy.

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

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

If you think about it, that's a bit tricky. If the code is directly in an `onclick` handler, then that's easy. But what is the popup opens in `setTimeout`?

Try this code:

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

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

…And this works in Firefox too:

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

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

Modern usage

As of now, we have many methods to load and show data on-page with JavaScript. But there are still situations when a popup works best.

For instance, many shops use online chats for consulting people. A visitor clicks on the button, it runs `window.open` and opens the popup with the chat.

Why a popup is good here, why not in-page?

1. A popup is a separate window with its own independent JavaScript environment. So a chat service doesn't need to integrate with scripts of the main shop site.
2. A popup is very simple to attach to a site, little to no overhead. It's only a small button, without additional scripts.
3. A popup may persist even if the user left the page. For example, a consult advices the user to visit the page of a new "Super-Cooler" goodie. The user goes there in the main window without leaving the chat.

window.open

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

url

An URL to load into the new window.

name

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

params

The configuration string for the new window. It contains settings, delimited by a comma. There must be no spaces in params, for instance: `width:200,height=100`.

Settings for `params`:

- Position:
  - `left/top` (numeric) – coordinates of the window top-left corner on the screen. There is a limitation: a new window cannot be positioned offscreen.
  - `width/height` (numeric) – width and height of a new window. There is a limit on minimal width/height, so it's impossible to create an invisible window.
- Window features:
  - `menubar` (yes/no) – shows or hides the browser menu on the new window.
  - `toolbar` (yes/no) – shows or hides the browser navigation bar (back, forward, reload etc) on the new window.

- `location` (yes/no) – shows or hides the URL field in the new window. FF and IE don’t allow to hide it by default.
- `status` (yes/no) – shows or hides the status bar. Again, most browsers force it to show.
- `resizable` (yes/no) – allows to disable the resize for the new window. Not recommended.
- `scrollbars` (yes/no) – allows to disable the scrollbars for the new window. Not recommended.

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

Example: a minimalistic window

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

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

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

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

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

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

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

Most browsers show the example above as required.

Rules for omitted settings:

- If there is no 3rd argument in the `open` call, or it is empty, then the default window parameters are used.
- If there is a string of params, but some yes/no features are omitted, then the omitted features are disabled, if the browser allows that. 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 a popup

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

In the example below, the contents of the new window is modified after loading.

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

newWindow.onload = function() {
  let html = `<div style="font-size:30px">Welcome!</div>`;
  newWindow.document.body.insertAdjacentHTML('afterbegin', html);
};
```

Please note that external `document` content is only accessible for windows from the same origin (the same protocol://domain:port).

For windows with URLs from another sites, we are able to change the location by assigning `newWindow.location=...`, but we can't read the location or access the content. That's for user safety, so that an evil page can't open a popup with `http://gmail.com` and read the data. We'll talk more about it later.

Accessing the opener window

A popup may access the "opener" window as well. A JavaScript in it may use `window.opener` to access the window that opened it. It is `null` for all windows except popups.

So both the main window and the popup have a reference to each other. They may modify each other freely assuming that they come from the same origin. If that's not so, then there are still means to communicate, to be covered in the next chapter Cross-window communication.

Closing a popup

If we don't need a popup any more, we can call `newWindow.close()` on it.

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()`.

The `newWindow.closed` 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 could close it, 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
};
```

Focus/blur on a popup

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

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

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

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

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

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

Still, there are some things that can be done.

For instance:

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

Summary

- A popup can be opened by the `open(url, name, params)` call. It returns the reference to the newly opened window.
- By default, browsers block `open` calls from the code outside of user actions. Usually a notification appears, so that a user may allow them.
- The popup may access the opener window using the `window.opener` property, so the two are connected.
- If the main window and the popup come from the same origin, they can freely read and modify each other. Otherwise, they can change location of each other and communicate using messages (to be covered).
- 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. Sometimes.
- 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`.

Also if we open a popup, a good practice is to notify the user about it. An icon with the opening window can help the visitor to survive the focus shift and keep both windows in mind.

Cross-window communication

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

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

Same Origin

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

These URLs all share the same origin:

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

These ones do not:

- `http://www.site.com` (another domain: `www.` matters)

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

The “Same Origin” policy states that:

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

Now let’s see how some examples. First, about pages that come from the same origin, and thus there are no limitations. And afterwards we’ll cover cross-window messaging that allows to work around the “Same Origin” policy.

Subdomains may be same-origin

There’s a small exclusion in the “Same Origin” policy.

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`), they can be treated as coming from the “same origin”.

To make it work, all such pages (including the one from `site.com`) should run the code:

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

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

Accessing an iframe contents

Our first example covers iframes. An `<iframe>` is a two-faced beast. From one side it’s a tag, just like `<script>` or `<img>`. From the other side it’s a window-in-window.

The embedded window has a separate `document` and `window` objects.

We can access them like using the properties:

- `iframe.contentWindow` is a reference to the window inside the `<iframe>`.

- `iframe.contentDocument` is a reference to the document inside the `<iframe>`.

When we access an embedded window, the browser checks if the iframe has the same origin. If that's not so then the access is denied (with exclusions noted above).

For instance, here's an `<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;

    try {
      // ...but not to the document inside it
      let doc = iframe.contentDocument;
    } catch(e) {
      alert(e); // Security Error (another origin)
    }

    // also we can't read the URL of the page in it
    try {
      alert(iframe.contentWindow.location);
    } catch(e) {
      alert(e); // Security Error
    }

    // ...but we can change it (and thus load something else into the iframe)!
    iframe.contentWindow.location = '/'; // works

    iframe.onload = null; // clear the handler, to run this code only once
  };
</script>
```

The code above shows errors for any operations except:

- Getting the reference to the inner window `iframe.contentWindow`
- Changing its `location`.

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

The `iframe.onload` event is actually the same as `iframe.contentWindow.onload`. It triggers when the embedded window fully loads with all resources.

…But `iframe.onload` is always available, while `iframe.contentWindow.onload` needs the same origin.

And now an example with the same origin. We can do anything with the embedded window:

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

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

Please wait until the iframe loads

When an iframe is created, it immediately has a document. But that document is different from the one that finally loads into it!

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>
```

That's actually a well-known pitfall for novice developers. We shouldn't work with the document immediately, because that's the *wrong document*. If we set any event handlers on it, they will be ignored.

…But the `onload` event triggers when the whole iframe with all resources is loaded. What if we want to act sooner, on `DOMContentLoaded` of the embedded document?

That's not possible if the iframe comes from another origin. But for the same origin we can try to catch the moment when a new document appears, and then setup necessary handlers, like this:

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

<script>
  let oldDoc = iframe.contentDocument;

  // every 100 ms check if the document is the new one
  let timer = setInterval(() => {
    if (iframe.contentDocument == oldDoc) return;
    // new document, let's set handlers
    iframe.contentDocument.addEventListener('DOMContentLoaded', () => {
      iframe.contentDocument.body.prepend('Hello, world!');
    });

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

Let me know in comments if you know a better solution here.

window.frames

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

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

For instance:

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

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

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

Navigation links are:

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

For instance:

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

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

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

The sandbox attribute

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

By default, for `<iframe sandbox src="...">` the “default set” of restrictions is applied to the iframe. But we can provide a space-separated list of “excluded” limitations as a value of the attribute, like this: `<iframe sandbox="allow-forms allow-popups">`. The listed limitations are not applied.

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:

http://plnkr.co/edit/GAhzx0j3JwAB1TMzwyxL?p=preview

Please note:

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

Cross-window messaging

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

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

The interface has two parts.

postMessage

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

Arguments:

`data`

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

`targetOrigin`

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

The `targetOrigin` is a safety measure. Remember, if the target window comes from another origin, we can't read it's `location`. So we can't be sure which site is open in the intended window right now: the user could navigate away.

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

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

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

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

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

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

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

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

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

onmessage

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

The event object has special properties:

`data`

The data from `postMessage`.

`origin`

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

`source`

The reference to the sender window. We can immediately `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 );
});
```

The full example:

http://plnkr.co/edit/ltrzlGvN8UPdpMtyxlI9?p=preview

There’s no delay

There’s totally no delay between `postMessage` and the `message` event. That happens synchronously, even faster than `setTimeout(...,0)`.

Summary

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

For popups we have two properties:

- `window.open` – opens a new window and returns a reference to it,
- `window.opener` – 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.

Exclusions 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 to talk with security checks:

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

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

The clickjacking attack

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

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

The idea

The idea is very simple.

Here’s how clickjacking was done with Facebook:

1. A visitor is lured to the evil page. It doesn’t matter how.
2. The page has a harmless-looking link on it (like “get rich now” or “click here, very funny”).
3. Over that link the evil page positions a transparent `<iframe>` with `src` from facebook.com, in such a way that the “Like” button is right above that link. Usually that’s done with `z-index`.
4. In attempting to click the link, the visitor in fact clicks the button.

The demo

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

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

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

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

<button>Click here!</button>

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

The full demo of the attack:

http://plnkr.co/edit/GQKK8Zc7DXT3KdV7tQUu?p=preview

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

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

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

http://plnkr.co/edit/aebDnhU3B7c6d2QN5Rhy?p=preview

All we need to attack – is to position the `<iframe>` on the evil page in such a way that the button is right over the link. That's usually possible with CSS.

Clickjacking is for clicks, not for keyboard

The attack only affects mouse actions.

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

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

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

Old-school defences (weak)

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

That looks like this:

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

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

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

Blocking top-navigation

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

The top page (belonging to the hacker) sets a handler to it, and when the `iframe` tries to change `top.location` the visitor gets a message asking them whether they want to leave.

Like this:

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

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

In action:

http://plnkr.co/edit/WCEMuiV3PmW1klyyf6FH?p=preview

Sandbox attribute

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

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

Here’s the code:

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

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

X-Frame-Options

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

It must be sent *by the server*: the browser will ignore it if found in a `<meta>` tag. So, `<meta http-equiv="X-Frame-Options"...>` won't do anything.

The header may have 3 values:

`DENY`

Never ever show the page inside a frame.

`SAMEORIGIN`

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

`ALLOW-FROM domain`

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

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

Showing with disabled functionality

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

So there are other solutions… For instance, we can "cover" the page with a `<div>` with `height: 100%; width: 100%;`, so that it intercepts all clicks. That `<div>` should disappear 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:

http://plnkr.co/edit/WuzXGKQamlVp1sE08svn?p=preview

Summary

Clickjacking is a way to “trick” users into clicking on a malicious 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.

Regular expressions

Regular expressions is a powerful way of doing search and replace in strings.

Patterns and flags

Regular expressions is a powerful way of searching and replacing inside a string.

In JavaScript regular expressions are implemented using objects of a built-in `RegExp` class and integrated with strings.

Please note that regular expressions vary between programming languages. In this tutorial we concentrate on JavaScript. Of course there’s a lot in common, but they are a somewhat different in Perl, Ruby, PHP etc.

Regular expressions

A regular expression (also “regexp”, or just “reg”) consists of a *pattern* and optional *flags*.

There are two syntaxes 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.

Usage

To search inside a string, we can use method search .

Here’s an example:

```
let str = "I love JavaScript!"; // will search here

let regexp = /love/;
alert( str.search(regexp) ); // 2
```

The `str.search` method looks for the pattern `/love/` and returns the position inside the string. As we might guess, `/love/` is the simplest possible pattern. What it does is a simple substring search.

The code above is the same as:

```
let str = "I love JavaScript!"; // will search here

let substr = 'love';
alert( str.search(substr) ); // 2
```

So searching for `/love/` is the same as searching for `"love"`.

But that’s only for now. Soon we’ll create more complex regular expressions with much searching more power.

Colors

From here on the color scheme is:

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

When to use `new RegExp`?

Normally we use the short syntax `/.../`. But it does not allow any variables insertions, so we must know the exact regexp at the time of writing the code.

From the other hand, `new RegExp` allows to construct a pattern dynamically from a string.

So we can figure out what we need to search and create `new RegExp` from it:

```
let search = prompt("What you want to search?", "love");
let regexp = new RegExp(search);

// find whatever the user wants
alert( "I love JavaScript".search(regexp));
```

Flags

Regular expressions may have flags that affect the search.

There are only 5 of them in JavaScript:

`i`

With this flag the search is case-insensitive: no difference between `A` and `a` (see the example below).

`g`

With this flag the search looks for all matches, without it – only the first one (we'll see uses in the next chapter).

`m`

Multiline mode (covered in the chapter Article "regexp-multiline" not found).

`u`

Enables full unicode support. The flag enables correct processing of surrogate pairs. More about that in the chapter The unicode flag.

`y`

Sticky mode (covered in the next chapter)

The "i" flag

The simplest flag is `i`.

An example with it:

```
let str = "I love JavaScript!";

alert( str.search(/LOVE/) ); // -1 (not found)
alert( str.search(/LOVE/i) ); // 2
```

1. The first search returns `-1` (not found), because the search is case-sensitive by default.
2. With the flag `/LOVE/i` the search found `love` at position 2.

So the `i` flag already makes regular expressions more powerful than a simple substring search. But there's so much more. We'll cover other flags and features in the next chapters.

Summary

- A regular expression consists of a pattern and optional flags: `g`, `i`, `m`, `u`, `y`.
- Without flags and special symbols that we'll study later, the search by a regexp is the same as a substring search.
- The method `str.search(regexp)` returns the index where the match is found or `-1` if there's no match.

Methods of RegExp and String

There are two sets of methods to deal with regular expressions.

1. First, regular expressions are objects of the built-in RegExp class, it provides many methods.
2. Besides that, there are methods in regular strings can work with regexps.

The structure is a bit messed up, so we'll first consider methods separately, and then – practical recipes for common tasks.

str.search(reg)

We've seen this method already. It returns the position of the first match or `-1` if none found:

```
let str = "A drop of ink may make a million think";

alert( str.search( /a/i ) ); // 0 (the first position)
```

The important limitation: `search` always looks for the first match.

We can't find next positions using `search`, there's just no syntax for that. But there are other methods that can.

str.match(reg), no "g" flag

The method `str.match` behavior varies depending on the `g` flag. First let's see the case without it.

Then `str.match(reg)` looks for the first match only.

The result is an array with that match and additional properties:

- `index` – the position of the match inside the string,
- `input` – the subject string.

For instance:

```
let str = "Fame is the thirst of youth";

let result = str.match( /fame/i );

alert( result[0] );    // Fame (the match)
alert( result.index ); // 0 (at the zero position)
alert( result.input ); // "Fame is the thirst of youth" (the string)
```

The array may have more than one element.

If a part of the pattern is delimited by parentheses `(...)`, then it becomes a separate element of the array.

For instance:

```
let str = "JavaScript is a programming language";

let result = str.match( /JAVA(SCRIPT)/i );

alert( result[0] ); // JavaScript (the whole match)
alert( result[1] ); // script (the part of the match that corresponds to the parenthes
alert( result.index ); // 0
alert( result.input ); // JavaScript is a programming language
```

Due to the `i` flag the search is case-insensitive, so it finds `JavaScript`. The part of the match that corresponds to `SCRIPT` becomes a separate array item.

We'll be back to parentheses later in the chapter Capturing groups. They are great for search-and-replace.

str.match(reg) with "g" flag

When there's a `"g"` flag, then `str.match` returns an array of all matches. There are no additional properties in that array, and parentheses do not create any elements.

For instance:

```
let str = "HO-Ho-ho!";

let result = str.match( /ho/ig );

alert( result ); // HO, Ho, ho (all matches, case-insensitive)
```

With parentheses nothing changes, here we go:

```
let str = "HO-Ho-ho!";

let result = str.match( /h(o)/ig );

alert( result ); // HO, Ho, ho
```

So, with `g` flag the `result` is a simple array of matches. No additional properties.

If we want to get information about match positions and use parentheses then we should use RegExp#exec method that we'll cover below.

⚠ If there are no matches, the call to `match` returns `null`

Please note, that’s important. If there were no matches, the result is not an empty array, but `null`.

Keep that in mind to evade pitfalls like this:

```
let str = "Hey-hey-hey!";

alert( str.match(/ho/gi).length ); // error! there's no length of null
```

str.split(regexp|substr, limit)

Splits the string using the regexp (or a substring) as a delimiter.

We already used `split` with strings, like this:

```
alert('12-34-56'.split('-')) // [12, 34, 56]
```

But we can also pass a regular expression:

```
alert('12-34-56'.split(/-/)) // [12, 34, 56]
```

str.replace(str|reg, str|func)

The swiss army knife for search and replace in strings.

The simplest use – search and replace a substring, like this:

```
// replace a dash by a colon
alert('12-34-56'.replace("-", ":")) // 12:34-56
```

When the first argument of `replace` is a string, it only looks for the first match.

To find all dashes, we need to use not the string `"-"`, but a regexp `/-/g`, with an obligatory `g` flag:

```
// replace all dashes by a colon
alert( '12-34-56'.replace( /-/g, ":" ) )  // 12:34:56
```

The second argument is a replacement string.

We can use special characters in it:

Symbol	Inserts
`$$`	`"$"`
`$&`	the whole match
`` $` ``	a part of the string before the match
`$'`	a part of the string after the match
`$n`	if `n` is a 1-2 digit number, then it means the contents of n-th parentheses counting from left to right

For instance let's use `$&` to replace all entries of `"John"` by `"Mr.John"`:

```
let str = "John Doe, John Smith and John Bull.";

// for each John - replace it with Mr. and then John
alert(str.replace(/John/g, 'Mr.$&'));
// "Mr.John Doe, Mr.John Smith and Mr.John Bull.";
```

Parentheses are very often used together with `$1`, `$2`, like this:

```
let str = "John Smith";

alert(str.replace(/(John) (Smith)/, '$2, $1')) // Smith, John
```

For situations that require "smart" replacements, the second argument can be a function.

It will be called for each match, and its result will be inserted as a replacement.

For instance:

```
let i = 0;

// replace each "ho" by the result of the function
alert("HO-Ho-ho".replace(/ho/gi, function() {
  return ++i;
})); // 1-2-3
```

In the example above the function just returns the next number every time, but usually the result is based on the match.

The function is called with arguments `func(str, p1, p2, ..., pn, offset, s)`:

1. `str` – the match,
2. `p1, p2, ..., pn` – contents of parentheses (if there are any),
3. `offset` – position of the match,
4. `s` – the source string.

If there are no parentheses in the regexp, then the function always has 3 arguments: `func(str, offset, s)`.

Let’s use it to show full information about matches:

```
// show and replace all matches
function replacer(str, offset, s) {
  alert(`Found ${str} at position ${offset} in string ${s}`);
  return str.toLowerCase();
}

let result = "HO-Ho-ho".replace(/ho/gi, replacer);
alert( 'Result: ' + result ); // Result: ho-ho-ho

// shows each match:
// Found HO at position 0 in string HO-Ho-ho
// Found Ho at position 3 in string HO-Ho-ho
// Found ho at position 6 in string HO-Ho-ho
```

In the example below there are two parentheses, so `replacer` is called with 5 arguments: `str` is the full match, then parentheses, and then `offset` and `s`:

```
function replacer(str, name, surname, offset, s) {
  // name is the first parentheses, surname is the second one
  return surname + ", " + name;
}

let str = "John Smith";

alert(str.replace(/(John) (Smith)/, replacer)) // Smith, John
```

Using a function gives us the ultimate replacement power, because it gets all the information about the match, has access to outer variables and can do everything.

regexp.test(str)

Let’s move on to the methods of `RegExp` class, that are callable on regexps themselves.

The `test` method looks for any match and returns `true/false` whether they found it.

So it’s basically the same as `str.search(reg) != -1`, for instance:

```
let str = "I love JavaScript";

// these two tests do the same
alert( /love/i.test(str) ); // true
alert( str.search(/love/i) != -1 ); // true
```

An example with the negative answer:

```
let str = "Bla-bla-bla";

alert( /love/i.test(str) ); // false
alert( str.search(/love/i) != -1 ); // false
```

regexp.exec(str)

We’ve already seen these searching methods:

- `search` – looks for the position of the match,
- `match` – if there’s no `g` flag, returns the first match with parentheses,
- `match` – if there’s a `g` flag – returns all matches, without separating parentheses.

The `regexp.exec` method is a bit harder to use, but it allows to search all matches with parentheses and positions.

It behaves differently depending on whether the regexp has the `g` flag.

- If there’s no `g`, then `regexp.exec(str)` returns the first match, exactly as `str.match(reg)`.
- If there’s `g`, then `regexp.exec(str)` returns the first match and *remembers* the position after it in `regexp.lastIndex` property. The next call starts to search from `regexp.lastIndex` and returns the next match. If there are no more matches then `regexp.exec` returns `null` and `regexp.lastIndex` is set to `0`.

As we can see, the method gives us nothing new if we use it without the `g` flag, because `str.match` does exactly the same.

But the `g` flag allows to get all matches with their positions and parentheses groups.

Here’s the example how subsequent `regexp.exec` calls return matches one by one:

```
let str = "A lot about JavaScript at https://javascript.info";

let regexp = /JAVA(SCRIPT)/ig;
```

```
// Look for the first match
let matchOne = regexp.exec(str);
alert( matchOne[0] ); // JavaScript
alert( matchOne[1] ); // script
alert( matchOne.index ); // 12 (the position of the match)
alert( matchOne.input ); // the same as str

alert( regexp.lastIndex ); // 22 (the position after the match)

// Look for the second match
let matchTwo = regexp.exec(str); // continue searching from regexp.lastIndex
alert( matchTwo[0] ); // javascript
alert( matchTwo[1] ); // script
alert( matchTwo.index ); // 34 (the position of the match)
alert( matchTwo.input ); // the same as str

alert( regexp.lastIndex ); // 44 (the position after the match)

// Look for the third match
let matchThree = regexp.exec(str); // continue searching from regexp.lastIndex
alert( matchThree ); // null (no match)

alert( regexp.lastIndex ); // 0 (reset)
```

As we can see, each `regexp.exec` call returns the match in a “full format”: as an array with parentheses, `index` and `input` properties.

The main use case for `regexp.exec` is to find all matches in a loop:

```
let str = 'A lot about JavaScript at https://javascript.info';

let regexp = /javascript/ig;

let result;

while (result = regexp.exec(str)) {
  alert( `Found ${result[0]} at ${result.index}` );
}
```

The loop continues until `regexp.exec` returns `null` that means “no more matches”.

Search from the given position

We can force `regexp.exec` to start searching from the given position by setting `lastIndex` manually:

```
let str = 'A lot about JavaScript at https://javascript.info';

let regexp = /javascript/ig;
regexp.lastIndex = 30;

alert( regexp.exec(str).index ); // 34, the search starts from the 30th position
```

The "y" flag

The `y` flag means that the search should find a match exactly at the position specified by the property `regexp.lastIndex` and only there.

In other words, normally the search is made in the whole string: `/javascript/` looks for “javascript” everywhere in the string.

But when a regexp has the `y` flag, then it only looks for the match at the position specified in `regexp.lastIndex` (`0` by default).

For instance:

```
let str = "I love JavaScript!";

let reg = /javascript/iy;

alert( reg.lastIndex ); // 0 (default)
alert( str.match(reg) ); // null, not found at position 0

reg.lastIndex = 7;
alert( str.match(reg) ); // JavaScript (right, that word starts at position 7)

// for any other reg.lastIndex the result is null
```

The regexp `/javascript/iy` can only be found if we set `reg.lastIndex=7`, because due to `y` flag the engine only tries to find it in the single place within a string – from the `reg.lastIndex` position.

So, what’s the point? Where do we apply that?

The reason is performance.

The `y` flag works great for parsers – programs that need to “read” the text and build in-memory syntax structure or perform actions from it. For that we move along the text and apply regular expressions to see what we have next: a string? A number? Something else?

The `y` flag allows to apply a regular expression (or many of them one-by-one) exactly at the given position and when we understand what's there, we can move on – step by step examining the text.

Without the flag the regexp engine always searches till the end of the text, that takes time, especially if the text is large. So our parser would be very slow. The `y` flag is exactly the right thing here.

Summary, recipes

Methods become much easier to understand if we separate them by their use in real-life tasks.

To search for the first match only:

- Find the position of the first match – `str.search(reg)`.
- Find the full match – `str.match(reg)`.
- Check if there's a match – `regexp.test(str)`.
- Find the match from the given position – `regexp.exec(str)`, set `regexp.lastIndex` to position.

To search for all matches:

- An array of matches – `str.match(reg)`, the regexp with `g` flag.
- Get all matches with full information about each one – `regexp.exec(str)` with `g` flag in the loop.

To search and replace:

- Replace with another string or a function result – `str.replace(reg, str|func)`

To split the string:

- `str.split(str|reg)`

We also covered two flags:

- The `g` flag to find all matches (global search),
- The `y` flag to search at exactly the given position inside the text.

Now we know the methods and can use regular expressions. But we need to learn their syntax, so let's move on.

Character classes

Consider a practical task – we have a phone number `"+7(903)-123-45-67"`, and we need to find all digits in that string. Other characters do not interest us.

A character class is a special notation that matches any symbol from the set.

For instance, there's a "digit" class. It's written as `\d`. We put it in the pattern, and during the search any digit matches it.

For instance, the regexp `/\d/` looks for a single digit:

```
let str = "+7(903)-123-45-67";

let reg = /\d/;

alert( str.match(reg) ); // 7
```

The regexp is not global in the example above, so it only looks for the first match.

Let's add the `g` flag to look for all digits:

```
let str = "+7(903)-123-45-67";

let reg = /\d/g;

alert( str.match(reg) ); // array of matches: 7,9,0,3,1,2,3,4,5,6,7
```

Most used classes: \d \s \w

That was a character class for digits. There are other character classes as well.

Most used are:

`\d` ("d" is from "digit")

A digit: a character from `0` to `9`.

`\s` ("s" is from "space")

A space symbol: that includes spaces, tabs, newlines.

`\w` ("w" is from "word")

A "wordly" character: either a letter of English alphabet or a digit or an underscore. Non-english letters (like cyrillic or hindi) do not belong to `\w`.

For instance, `\d\s\w` means a digit followed by a space character followed by a wordly character, like `"1 Z"`.

A regexp may contain both regular symbols and character classes.

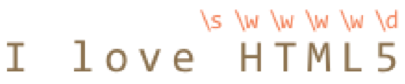
For instance, `CSS\d` matches a string `CSS` with a digit after it:

```
let str = "CSS4 is cool";
let reg = /CSS\d/

alert( str.match(reg) ); // CSS4
```

Also we can use many character classes:

```
alert( "I love HTML5!".match(/\s\w\w\w\w\d/) ); // 'HTML5'
```

The match (each character class corresponds to one result character):



Word boundary: \b

The word boundary `\b` – is a special character class.

It does not denote a character, but rather a boundary between characters.

For instance, `\bJava\b` matches `Java` in the string `Hello, Java!`, but not in the script `Hello, JavaScript!`.

```
alert( "Hello, Java!".match(/\bJava\b/) ); // Java
alert( "Hello, JavaScript!".match(/\bJava\b/) ); // null
```

The boundary has “zero width” in a sense that usually a character class means a character in the result (like a wordly or a digit), but not in this case.

The boundary is a test.

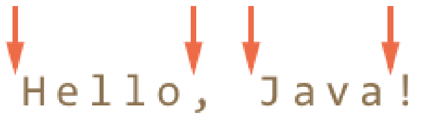
When regular expression engine is doing the search, it’s moving along the string in an attempt to find the match. At each string position it tries to find the pattern.

When the pattern contains `\b`, it tests that the position in string fits one of the conditions:

- String start, and the first string character is `\w`.
- String end, and the last string character is `\w`.

- Inside the string: from one side is `\w`, from the other side – not `\w`.

For instance, in the string `Hello, Java!` the following positions match `\b`:



So it matches `\bHello\b` and `\bJava\b`, but not `\bHell\b` (because there's no word boundary after `l`) and not `Java!\b` (because the exclamation sign is not a wordly character, so there's no word boundary after it).

```
alert( "Hello, Java!".match(/\bHello\b/) ); // Hello
alert( "Hello, Java!".match(/\bJava\b/) );  // Java
alert( "Hello, Java!".match(/\bHell\b/) );  // null
alert( "Hello, Java!".match(/\bJava!\b/) ); // null
```

Once again let's note that `\b` makes the searching engine to test for the boundary, so that `Java\b` finds `Java` only when followed by a word boundary, but it does not add a letter to the result.

Usually we use `\b` to find standalone English words. So that if we want `"Java"` language then `\bJava\b` finds exactly a standalone word and ignores it when it's a part of `"JavaScript"`.

Another example: a regexp `\b\d\d\b` looks for standalone two-digit numbers. In other words, it requires that before and after `\d\d` must be a symbol different from `\w` (or beginning/end of the string).

```
alert( "1 23 456 78".match(/\b\d\d\b/g) ); // 23,78
```

Word boundary doesn't work for non-English alphabets

The word boundary check `\b` tests for a boundary between `\w` and something else. But `\w` means an English letter (or a digit or an underscore), so the test won't work for other characters (like cyrillic or hieroglyphs).

Reverse classes

For every character class there exists a "reverse class", denoted with the same letter, but uppercased.

The "reverse" means that it matches all other characters, for instance:

`\D`

Non-digit: any character except `\d`, for instance a letter.

`\S`

Non-space: any character except `\s`, for instance a letter.

`\W`

Non-wordly character: anything but `\w`.

`\B`

Non-boundary: a test reverse to `\b`.

In the beginning of the chapter we saw how to get all digits from the phone `+7(903)-123-45-67`. Let's get a "pure" phone number from the string:

```
let str = "+7(903)-123-45-67";

alert( str.match(/\d/g).join('') ); // 79031234567
```

An alternative way would be to find non-digits and remove them from the string:

```
let str = "+7(903)-123-45-67";

alert( str.replace(/\D/g, "") ); // 79031234567
```

Spaces are regular characters

Please note that regular expressions may include spaces. They are treated like regular characters.

Usually we pay little attention to spaces. For us strings `1-5` and `1 - 5` are nearly identical.

But if a regexp does not take spaces into account, it won' work.

Let's try to find digits separated by a dash:

```
alert( "1 - 5".match(/\d-\d/) ); // null, no match!
```

Here we fix it by adding spaces into the regexp:

```
alert( "1 - 5".match(/\d - \d/) ); // 1 - 5, now it works
```

Of course, spaces are needed only if we look for them. Extra spaces (just like any other extra characters) may prevent a match:

```
alert( "1-5".match(/\d - \d/) ); // null, because the string 1-5 has no spaces
```

In other words, in a regular expression all characters matter. Spaces too.

A dot is any character

The dot `"."` is a special character class that matches *any character except a newline*.

For instance:

```
alert( "Z".match(/./) ); // Z
```

Or in the middle of a regexp:

```
let reg = /CS.4/;

alert( "CSS4".match(reg) ); // CSS4
alert( "CS-4".match(reg) ); // CS-4
alert( "CS 4".match(reg) ); // CS 4 (space is also a character)
```

Please note that the dot means “any character”, but not the “absense of a character”. There must be a character to match it:

```
alert( "CS4".match(/CS.4/) ); // null, no match because there's no character for the d
```

Summary

We covered character classes:

- `\d` – digits.
- `\D` – non-digits.
- `\s` – space symbols, tabs, newlines.
- `\S` – all but `\s`.
- `\w` – English letters, digits, underscore `'_'`.
- `\W` – all but `\w`.
- `'.'` – any character except a newline.

If we want to search for a character that has a special meaning like a backslash or a dot, then we should escape it with a backslash: `\.`

Please note that a regexp may also contain string special characters such as a newline `\n`. There's no conflict with character classes, because other letters are used for them.

✓ Tasks

Find the time

The time has a format: `hours:minutes`. Both hours and minutes has two digits, like `09:00`.

Make a regexp to find time in the string: `Breakfast at 09:00 in the room 123:456.`

P.S. In this task there's no need to check time correctness yet, so `25:99` can also be a valid result. P.P.S. The regexp shouldn't match `123:456`.

To solution

Escaping, special characters

As we've seen, a backslash `"\"` is used to denote character classes. So it's a special character.

There are other special characters as well, that have special meaning in a regexp. They are used to do more powerful searches.

Here's a full list of them: `[ \ ^ $ . | ? * + ( )`.

Don't try to remember it – when we deal with each of them separately, you'll know it by heart automatically.

Escaping

To use a special character as a regular one, prepend it with a backslash.

That's also called "escaping a character".

For instance, we need to find a dot `'.'`. In a regular expression a dot means "any character except a newline", so if we really mean "a dot", let's put a backslash before it: `\.`.

```
alert( "Chapter 5.1".match(/\d\.\d/) ); // 5.1
```

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 `\`, then we should double it:

```
alert( "1\\2".match(/\\/) ); // '\'
```

A slash

The 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(/\//) ); // '/'
```

From the other hand, the alternative `new RegExp` syntaxes does not require escaping it:

```
alert( "/".match(new RegExp("/")) ); // '/'
```

new RegExp

If we are creating a regular expression with `new RegExp`, then we need to do some more escaping.

For instance, consider this:

```
let reg = new RegExp("\d\.\d");

alert( "Chapter 5.1".match(reg) ); // null
```

It doesn't work, but why?

The reason is string escaping rules. Look here:

```
alert("\d\.\d"); // d.d
```

Backslashes are used for escaping inside a string and string-specific special characters like `\n`. The quotes "consume" and interpret them, for instance:

- `\n` – becomes a newline character,
- `\u1234` – becomes the Unicode character with such code,
- …And when there's no special meaning: like `\d` or `\z`, then the backslash is simply removed.

So the call to `new RegExp` gets a string without backslashes.

To fix it, we need to double backslashes, because quotes turn `\\` into `\`:

```
let regStr = "\\d\\.\\d";
alert(regStr); // \d\.\d (correct now)

let reg = new RegExp(regStr);

alert( "Chapter 5.1".match(reg) ); // 5.1
```

Sets and ranges [...]

Several characters or character classes inside square brackets `[…]` mean to "search for any character among given".

Sets

For instance, `[eao]` means any of the 3 characters: `'a'`, `'e'`, or `'o'`.

That's called a *set*. Sets can be used in a regexp along with regular characters:

```
// find [t or m], and then "op"
alert( "Mop top".match(/[tm]op/gi) ); // "Mop", "top"
```

Please note that although there are multiple characters in the set, they correspond to exactly one character in the match.

So the example above gives no matches:

```
// find "V", then [o or i], then "la"
alert( "Voila".match(/V[oi]la/) ); // null, no matches
```

The pattern assumes:

- `V`,

- then *one* of the letters `[oi]`,
- then `la`.

So there would be a match for `Vola` or `Vila`.

Ranges

Square brackets may also contain *character ranges*.

For instance, `[a-z]` is a character in range from `a` to `z`, and `[0-5]` is a digit from `0` to `5`.

In the example below we're searching for `"x"` followed by two digits or letters from `A` to `F`:

```
alert( "Exception 0xAF".match(/x[0-9A-F][0-9A-F]/g) ); // xAF
```

Please note that in the word `Exception` there's a substring `xce`. It didn't match the pattern, because the letters are lowercase, while in the set `[0-9A-F]` they are uppercase.

If we want to find it too, then we can add a range `a-f`: `[0-9A-Fa-f]`. The `i` flag would allow lowercase too.

Character classes are shorthands for certain character sets.

For instance:

- **\d** – is the same as `[0-9]`,
- **\w** – is the same as `[a-zA-Z0-9_]`,
- **\s** – is the same as `[\t\n\v\f\r ]` plus few other unicode space characters.

We can use character classes inside `[…]` as well.

For instance, we want to match all wordly characters or a dash, for words like "twenty-third". We can't do it with `\w+`, because `\w` class does not include a dash. But we can use `[\w-]`.

We also can use a combination of classes to cover every possible character, like `[\s\S]`. That matches spaces or non-spaces – any character. That's wider than a dot `"."`, because the dot matches any character except a newline.

Excluding ranges

Besides normal ranges, there are "excluding" ranges that look like `[^…]`.

They are denoted by a caret character `^` at the start and match any character *except the given ones*.

For instance:

- `[^aeyo]` – any character except `'a'`, `'e'`, `'y'` or `'o'`.
- `[^0-9]` – any character except a digit, the same as `\D`.
- `[^\s]` – any non-space character, same as `\S`.

The example below looks for any characters except letters, digits and spaces:

```
alert( "alice15@gmail.com".match(/[^\d\sA-Z]/gi) ); // @ and .
```

No escaping in […]

Usually when we want to find exactly the dot character, we need to escape it like `\.`. And if we need a backslash, then we use `\\`.

In square brackets the vast majority of special characters can be used without escaping:

- A dot `'.'`.
- A plus `'+'`.
- Parentheses `'( )'`.
- Dash `'-'` in the beginning or the end (where it does not define a range).
- A caret `'^'` if not in the beginning (where it means exclusion).
- And the opening square bracket `'['`.

In other words, all special characters are allowed except where they mean something for square brackets.

A dot `"."` inside square brackets means just a dot. The pattern `[.,]` would look for one of characters: either a dot or a comma.

In the example below the regexp `[-().^+]` looks for one of the characters `-().^+`:

```
// No need to escape
let reg = /[-().^+]/g;

alert( "1 + 2 - 3".match(reg) ); // Matches +, -
```

…But if you decide to escape them “just in case”, then there would be no harm:

```
// Escaped everything
let reg = /[\-\(\)\.\^\+]/g;

alert( "1 + 2 - 3".match(reg) ); // also works: +, -
```

✓ Tasks

Java[^script]

We have a regexp `/Java[^script]/`.

Does it match anything in the string `Java`? In the string `JavaScript`?

To solution

Find the time as hh:mm or hh-mm

The time can be in the format `hours:minutes` or `hours-minutes`. Both hours and minutes have 2 digits: `09:00` or `21-30`.

Write a regexp to find time:

```
let reg = /your regexp/g;
alert( "Breakfast at 09:00. Dinner at 21-30".match(reg) ); // 09:00, 21-30
```

P.S. In this task we assume that the time is always correct, there's no need to filter out bad strings like "45:67". Later we'll deal with that too.

To solution

The unicode flag

The unicode flag `/.../u` enables the correct support of surrogate pairs.

Surrogate pairs are explained in the chapter Strings.

Let's briefly remind them here. In short, normally characters are encoded with 2 bytes. That gives us 65536 characters maximum. But there are more characters in the world.

So certain rare characters are encoded with 4 bytes, like `𝒳` (mathematical X) or `😄` (a smile).

Here are the unicode values to compare:

Character	Unicode	Bytes
`a`	0x0061	2
`≈`	0x2248	2
`𝒳`	0x1d4b3	4
`𝒴`	0x1d4b4	4
`😄`	0x1f604	4

So characters like `a` and `≈` occupy 2 bytes, and those rare ones take 4.

The unicode is made in such a way that the 4-byte characters only have a meaning as a whole.

In the past JavaScript did not know about that, and many string methods still have problems. For instance, `length` thinks that here are two characters:

```
alert('😄'.length); // 2
alert('𝒳'.length); // 2
```

…But we can see that there's only one, right? The point is that `length` treats 4 bytes as two 2-byte characters. That's incorrect, because they must be considered only together (so-called "surrogate pair").

Normally, regular expressions also treat "long characters" as two 2-byte ones.

That leads to odd results, for instance let's try to find `[𝒳𝒴]` in the string `𝒳`:

```
alert( '𝒳'.match(/[𝒳𝒴]/) ); // odd result
```

The result would be wrong, because by default the regexp engine does not understand surrogate pairs. It thinks that `[𝒳𝒴]` are not two, but four characters: the left half of `𝒳` `(1)`, the right half of `𝒳` `(2)`, the left half of `𝒴` `(3)`, the right half of `𝒴` `(4)`.

So it finds the left half of `𝒳` in the string `𝒳`, not the whole symbol.

In other words, the search works like `'12'.match(/[1234]/)` – the `1` is returned (left half of `𝒳`).

The `/.../u` flag fixes that. It enables surrogate pairs in the regexp engine, so the result is correct:

```
alert( '𝒳'.match(/[𝒳𝒴]/u) ); // 𝒳
```

There's an error that may happen if we forget the flag:

```
'𝒳'.match(/[𝒳-𝒴]/); // SyntaxError: invalid range in character class
```

Here the regexp `[𝒳-𝒴]` is treated as `[12-34]` (where `2` is the right part of `𝒳` and `3` is the left part of `𝒴`), and the range between two halves `2` and `3` is unacceptable.

Using the flag would make it work right:

```
alert( '𝒴'.match(/[𝒳-𝒵]/u) ); // 𝒴
```

To finalize, let's note that if we do not deal with surrogate pairs, then the flag does nothing for us. But in the modern world we often meet them.

Quantifiers +, *, ? and {n}

Let's say we have a string like `+7(903)-123-45-67` and want to find all numbers in it. But unlike before, we are interested in not digits, but full numbers: `7, 903, 123, 45, 67`.

A number is a sequence of 1 or more digits `\d`. The instrument to say how many we need is called *quantifiers*.

Quantity {n}

The most obvious quantifier is a number in figure quotes: `{n}`. A quantifier is put after a character (or a character class and so on) and specifies exactly how many we need.

It also has advanced forms, here we go with examples:

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 count from-to: `{3,5}`

To find numbers from 3 to 5 digits we can put the limits into figure brackets: `\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 numbers of `3` and more digits:

```
alert( "I'm not 12, but 345678 years old".match(/\d{3,}/) ); // "345678"
```

In case with the string `+7(903)-123-45-67` we need numbers: one or more digits in a row. That is `\d{1,}`:

```
let str = "+7(903)-123-45-67";

let numbers = str.match(/\d{1,}/g);

alert(numbers); // 7,903,123,45,67
```

Shorthands

Most often needed quantifiers have shorthands:

`+`

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 it can find `or` in the word `color` and `our` in `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.

The example below looks for a digit followed by any number of zeroes:

```
alert( "100 10 1".match(/\d0*/g) ); // 100, 10, 1
```

Compare it with `'+'` (one or more):

```
alert( "100 10 1".match(/\d0+/g) ); // 100, 10
```

More examples

Quantifiers are used very often. They are one of the main “building blocks” for complex regular expressions, so let’s see more examples.

Regexp “decimal fraction” (a number with a floating point): `\d+\.\d+`

In action:

```
alert( "0 1 12.345 7890".match(/\d+\.\d+/g) ); // 12.345
```

Regexp “open HTML-tag without attributes”, like `<span>` or `<p>`: `/<[a-z]+>/i`

In action:

```
alert( "<body> ... </body>".match(/<[a-z]+>/gi) ); // <body>
```

We look for character `'<'` followed by one or more English letters, and then `'>'`.

Regexp “open HTML-tag without attributes” (improved): `/<[a-z][a-z0-9]*>/i`

Better regexp: according to the standard, HTML tag name may have a digit at any position except the first one, like `<h1>`.

```
alert( "<h1>Hi!</h1>".match(/<[a-z][a-z0-9]*>/gi) ); // <h1>
```

Regexp “opening or closing HTML-tag without attributes”: `/<\/?[a-z][a-z0-9]*>/i`

We added an optional slash `/?` before the tag. Had to escape it with a backslash, otherwise JavaScript would think it is the pattern end.

```
alert( "<h1>Hi!</h1>".match(/<\/?[a-z][a-z0-9]*>/gi) ); // <h1>, </h1>
```

More precise means 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, HTML tags could use a simpler regexp: `<\w+>`.

Because `\w` means any English letter or a digit or `'_'`, the regexp also matches non-tags, for instance `<_>`. But it's much simpler than `<[a-z][a-z0-9]*>`.

Are we ok with `<\w+>` or we need `<[a-z][a-z0-9]*>`?

In real life both variants are acceptable. Depends on how tolerant we can be to "extra" matches and whether it's difficult or not to filter them out by other means.

Tasks

How to find an ellipsis "..." ?

importance: 5

Create a regexp to find ellipsis: 3 (or more?) dots in a row.

Check it:

```
let reg = /your regexp/g;
alert( "Hello!... How goes?.....".match(reg) ); // ..., .....
```

To solution

Regexp for HTML colors

Create a regexp to search HTML-colors written as `#ABCDEF`: first `#` and then 6 hexadimal characters.

An example of use:

```
let reg = /...your regexp.../

let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2 #12345678";

alert( str.match(reg) )  // #121212,#AA00ef
```

P.S. In this task we do not need other color formats like `#123` or `rgb(1,2,3)` etc.

To solution

Greedy and lazy quantifiers

Quantifiers are very simple from the first sight, but in fact they can be tricky.

We should understand how the search works very well if we plan to look for something more complex than `/\d+/`.

Let’s take the following task as an example.

We have a text and need to replace all quotes `"..."` with guillemet marks: `«...»`. They are preferred for typography in many countries.

For instance: `"Hello, world"` should become `«Hello, world»`.

Some countries prefer `„Witam, świat!”` (Polish) or even `「你好，世界」` (Chinese) quotes. For different locales we can choose different replacements, but that all works the same, so let’s start with `«...»`.

To make replacements we first need to find all quoted substrings.

The regular expression can look like this: `/".+"/g`. That is: we look for a quote followed by one or more characters, and then another quote.

…But if we try to apply it, even in such a simple case…

```
let reg = /".+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(reg) ); // "witch" and her "broom"
```

…We can see that it works not as intended!

Instead of finding two matches `"witch"` and `"broom"`, it finds one: `"witch" and her "broom"`.

That can be described as “greediness is the cause of all evil”.

Greedy search

To find a match, the regular expression engine uses the following algorithm:

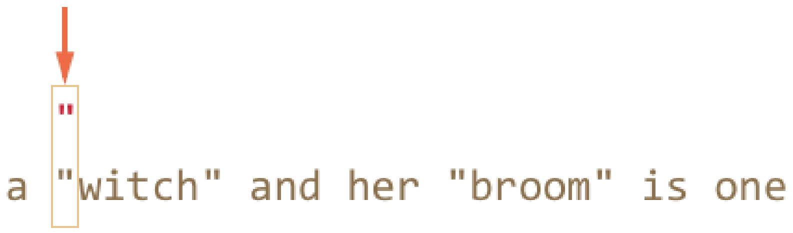
- For every position in the string
  - Match the pattern at that position.
  - If there’s no match, go to the next position.

These common words do not make it obvious why the regexp fails, so let's elaborate how the search works for the pattern `".+"`.

1. The first pattern character is a quote `"`.

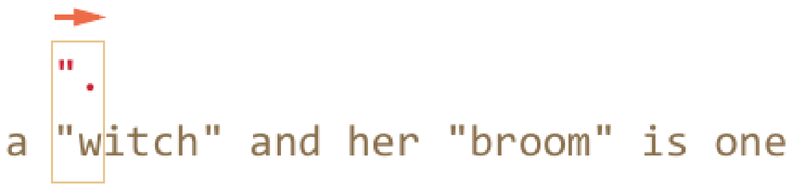
   The regular expression engine tries to find it at the zero position of the source string `a "witch" and her "broom" is one`, but there's `a` there, so there's immediately no match.

   Then it advances: goes to the next positions in the source string and tries to find the first character of the pattern there, and finally finds the quote at the 3rd position:

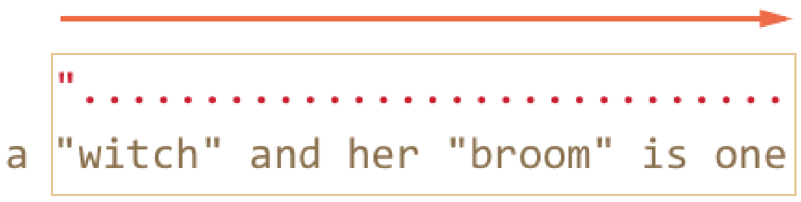
2. The quote is detected, and then the engine tries to find a match for the rest of the pattern. It tries to see if the rest of the subject string conforms to `.+"`.

   In our case the next pattern character is `.` (a dot). It denotes "any character except a newline", so the next string letter `'w'` fits:

3. Then the dot repeats because of the quantifier `.+`. The regular expression engine builds the match by taking characters one by one while it is possible.

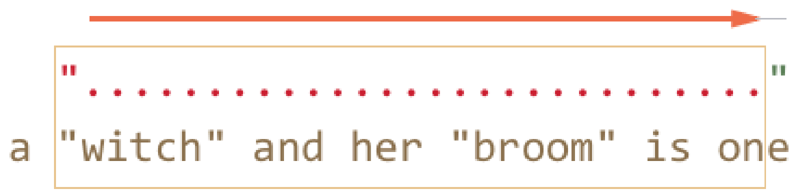
   …When it becomes impossible? All characters match the dot, so it only stops when it reaches the end of the string:

4. Now the engine finished repeating for `.+` and tries to find the next character of the pattern. It's the quote `"`. But there's a problem: the string has finished, there are no more characters!

   The regular expression engine understands that it took too many `.+` and starts to *backtrack*.

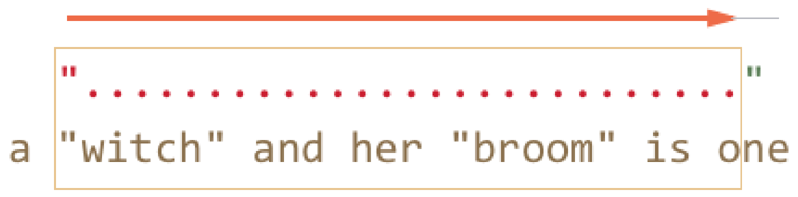
   In other words, it shortens the match for the quantifier by one character:



Now it assumes that `.+` ends one character before the end and tries to match the rest of the pattern from that position.

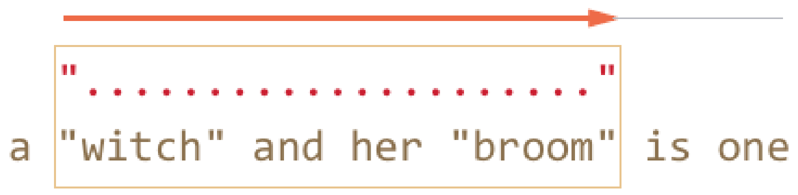
If there were a quote there, then that would be the end, but the last character is `'e'`, so there's no match.

5. …So the engine decreases the number of repetitions of `.+` by one more character:



The quote `'"'` does not match `'n'`.

6. The engine keep backtracking: it decreases the count of repetition for `'.'` until the rest of the pattern (in our case `'"'`) matches:



7. The match is complete.

8. So the first match is `"witch" and her "broom"`. The further search starts where the first match ends, but there are no more quotes in the rest of the string `is one`, so no more results.

That's probably not what we expected, but that's how it works.

In the greedy mode (by default) the quantifier is repeated as many times as possible.

The regexp engine tries to fetch as many characters as it can by `.+`, and then shortens that one by one.

For our task we want another thing. That's what the lazy quantifier mode is for.

Lazy mode

The lazy mode of quantifier is an opposite to the greedy mode. It means: "repeat minimal number of times".

We can enable it by putting a question mark `'?'` after the quantifier, so that it becomes `*?` or `+?` or even `??` for `'?'`.

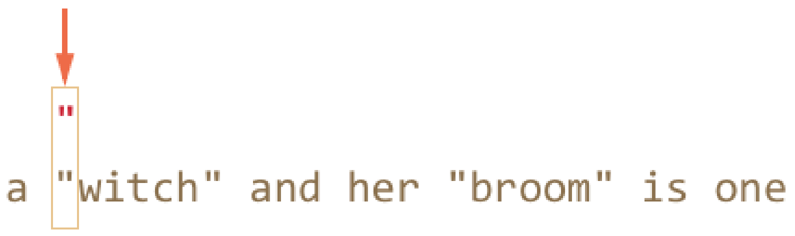
To make things clear: usually a question mark `?` is a quantifier by itself (zero or one), but if added *after another quantifier (or even itself)* it gets another meaning – it switches the matching mode from greedy to lazy.

The regexp `/".+?"/g` works as intended: it finds `"witch"` and `"broom"`:

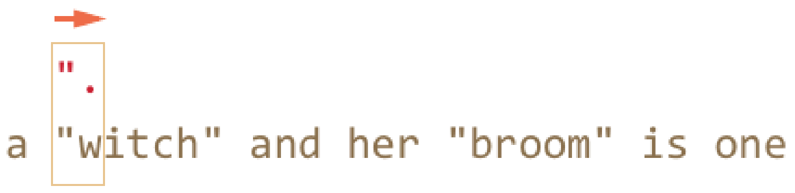
```
let reg = /".+?"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(reg) ); // witch, broom
```

To clearly understand the change, let's trace the search step by step.

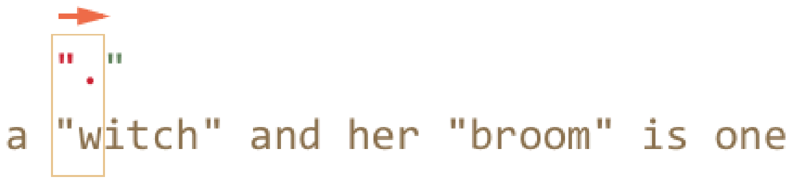
1. The first step is the same: it finds the pattern start `'"'` at the 3rd position:



2. The next step is also similar: the engine finds a match for the dot `'.'`:

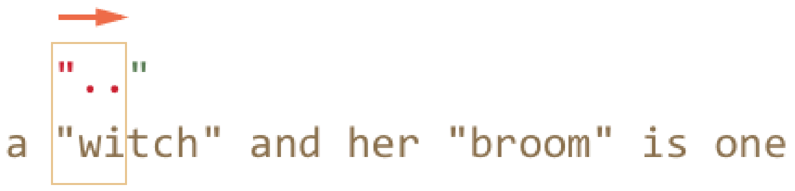


3. And now the search goes differently. Because we have a lazy mode for `+?`, the engine doesn't try to match a dot one more time, but stops and tries to match the rest of the pattern `'"'` right now:



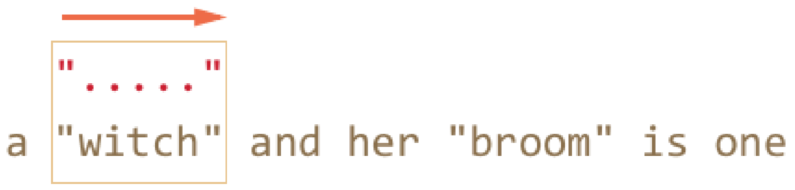
   If there were a quote there, then the search would end, but there's `'i'`, so there's no match.

4. Then the regular expression engine increases the number of repetitions for the dot and tries one more time:

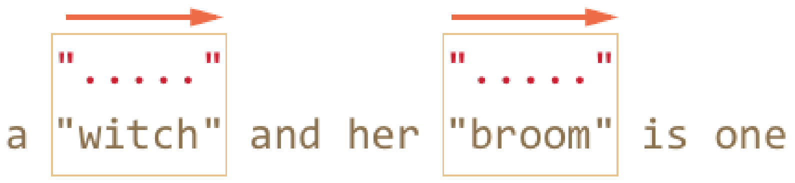


Failure again. Then the number of repetitions is increased again and again…

5. …Till the match for the rest of the pattern is found:



6. 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+?/g) ); // 123 4
```

1. The pattern `\d+` tries to match as many numbers as it can (greedy mode), so it finds `123` and stops, because the next character is a space `' '`.
2. Then there's a space in pattern, it matches.
3. Then there's `\d+?`. The quantifier is in lazy mode, so it finds one digit `4` and tries to check if the rest of the pattern matches from there.

   …But there's nothing in the pattern after `\d+?`.

   The lazy mode doesn't repeat anything without a need. The pattern finished, so we're done. We have a match `123 4`.
4. The next search starts from the character `5`.

Optimizations

Modern regular expression engines can optimize internal algorithms to work faster. So they may work a bit different from the described algorithm.

But to understand how regular expressions work and to build regular expressions, we don't need to know about that. They are only used internally to optimize things.

Complex regular expressions are hard to optimize, so the search may work exactly as described as well.

Alternative approach

With regexps, there's often more than one way to do the same thing.

In our case we can find quoted strings without lazy mode using the regexp `"[^"]+"`:

```
let reg = /"[^"]+"/g;

let str = 'a "witch" and her "broom" is one';

alert( str.match(reg) ); // witch, broom
```

The regexp `"[^"]+"` gives correct results, because it looks for a quote `'"'` followed by one or more non-quotes `[^"]`, and then the closing quote.

When the regexp engine looks for `[^"]+` it stops the repetitions when it meets the closing quote, and we're done.

Please note, that this logic does not replace lazy quantifiers!

It is just different. There are times when we need one or another.

Let's see one more 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 reg = /<a href=".*" class="doc">/g;

// Works!
alert( str.match(reg) ); // <a href="link" class="doc">
```

…But what if there are many links in the text?

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let reg = /<a href=".*" class="doc">/g;

// Whoops! Two links in one match!
alert( str.match(reg) ); // <a href="link1" class="doc">... <a href="link2" class="doc
```

Now the result is wrong for the same reason as our “witches” example. The quantifier `.*` took too many characters.

The match looks like this:

```
<a href="....................................." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

Let’s modify the pattern by making the quantifier `.*?` lazy:

```
let str = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let reg = /<a href=".*?" class="doc">/g;

// Works!
alert( str.match(reg) ); // <a href="link1" class="doc">, <a href="link2" class="doc">
```

Now it works, there are two matches:

```
<a href="....." class="doc">    <a href="....." class="doc">
<a href="link1" class="doc">... <a href="link2" class="doc">
```

Why it works – should be obvious after all explanations above. So let’s not stop on the details, but try one more text:

```
let str = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let reg = /<a href=".*?" class="doc">/g;

// Wrong match!
alert( str.match(reg) ); // <a href="link1" class="wrong">... <p style="" class="doc">
```

We can see that the regexp matched not just a link, but also a lot of text after it, including `<p...>`.

Why it happens?

1. First the regexp finds a link start `<a href="`.
2. Then it looks for `.*?`, we take one character, then check if there's a match for the rest of the pattern, then take another one…

   The quantifier `.*?` consumes characters until it meets `class="doc">`.

   …And where can it find it? If we look at the text, then we can see that the only `class="doc">` is beyond the link, in the tag `<p>`.
3. So we have match:

   ```
   <a href="..................................." class="doc">
   <a href="link1" class="wrong">... <p style="" class="doc">
   ```

So the laziness did not work for us here.

We need the pattern to look for `<a href="...something..." class="doc">`, but both greedy and lazy variants have problems.

The correct variant would be: `href="[^"]*"`. It will take all characters inside the `href` attribute till the nearest quote, just what we need.

A working example:

```
let str1 = '...<a href="link1" class="wrong">... <p style="" class="doc">...';
let str2 = '...<a href="link1" class="doc">... <a href="link2" class="doc">...';
let reg = /<a href="[^"]*" class="doc">/g;

// Works!
alert( str1.match(reg) ); // null, no matches, that's correct
alert( str2.match(reg) ); // <a href="link1" class="doc">, <a href="link2" class="doc"
```

Summary

Quantifiers have two modes of work:

Greedy

By default the regular expression engine tries to repeat the quantifier as many times as possible. For instance, `\d+` consumes all possible digits. When it becomes impossible to consume more (no more digits or string end), then it continues to match the rest of the pattern. If there's no match then it decreases the number of repetitions (backtracks) and tries again.

Lazy

Enabled by the question mark `?` after the quantifier. The regexp engine tries to match the rest of the pattern before each repetition of the quantifier.

As we've seen, the lazy mode is not a "panacea" from the greedy search. An alternative is a "fine-tuned" greedy search, with exclusions. Soon we'll see more examples of it.

✓ Tasks

A match for /d+? d+?/

What's the match here?

```
"123 456".match(/\d+? \d+?/g) ); // ?
```

To solution

Find HTML comments

Find all HTML comments in the text:

```
let reg = /your regexp/g;

let str = `... <!-- My -- comment
 test --> ..  <!----> ..
`;

alert( str.match(reg) ); // '<!-- My -- comment \n test -->', '<!---->'
```

To solution

Find HTML tags

Create a regular expression to find all (opening and closing) HTML tags with their attributes.

An example of use:

```
let reg = /your regexp/g;

let str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(reg) ); // '<a href="/">', '<input type="radio" checked>', '<b>'
```

Let's assume that may not contain `<` and `>` inside (in quotes too), that simplifies things a bit.

To solution

Capturing groups

A part of a pattern can be enclosed in parentheses `(...)`. This is called a “capturing group”.

That has two effects:

1. It allows to place a part of the match into a separate array item when using String#match or RegExp#exec methods.
2. If we put a quantifier after the parentheses, it applies to the parentheses as a whole, not the last character.

Example

In the example below the pattern `(go)+` finds one or more `'go'`:

```
alert( 'Gogogo now!'.match(/(go)+/i) ); // "Gogogo"
```

Without parentheses, the pattern `/go+/` means `g`, followed by `o` repeated one or more times. For instance, `goooo` or `gooooooooo`.

Parentheses group the word `(go)` together.

Let’s make something more complex – a regexp to match an email.

Examples of emails:

```
my@mail.com
john.smith@site.com.uk
```

The pattern: `[-.\w]+@([\w-]+\.)+[\w-]{2,20}`.

- The first part before `@` may include any alphanumeric word characters, a dot and a dash `[-.\w]+`, like `john.smith`.
- Then `@`
- And then the domain and maybe a second-level domain like `site.com` or with subdomains like `host.site.com.uk`. We can match it as “a word followed by a dot” repeated one or more times for subdomains: `mail.` or `site.com.`, and then “a word” for the last part: `.com` or `.uk`.

  The word followed by a dot is `(\w+\.)+` (repeated). The last word should not have a dot at the end, so it’s just `\w{2,20}`. The quantifier `{2,20}` limits the length, because domain zones are like `.uk` or `.com` or `.museum`, but can’t be longer than 20 characters.

So the domain pattern is `(\w+\.)+\w{2,20}`. Now we replace `\w` with `[\w-]`, because dashes are also allowed in domains, and we get the final result.

That regexp is not perfect, but usually works. It's short and good enough to fix errors or occasional mistypes.

For instance, here we can find all emails in the string:

```
let reg = /[-.\w]+@([\w-]+\.)+[\w-]{2,20}/g;

alert("my@mail.com @ his@site.com.uk".match(reg)); // my@mail.com,his@site.com.uk
```

Contents of parentheses

Parentheses are numbered from left to right. The search engine remembers the content of each and allows to reference it in the pattern or in the replacement string.

For instance, we can find an HTML-tag using a (simplified) pattern `<.*?>`. Usually we'd want to do something with the result after it.

If we enclose the inner contents of `<...>` into parentheses, then we can access it like this:

```
let str = '<h1>Hello, world!</h1>';
let reg = /<(.*?)>/;

alert( str.match(reg) ); // Array: ["<h1>", "h1"]
```

The call to String#match returns groups only if the regexp has no `/.../g` flag.

If we need all matches with their groups then we can use RegExp#exec method as described in Methods of RegExp and String:

```
let str = '<h1>Hello, world!</h1>';

// two matches: opening <h1> and closing </h1> tags
let reg = /<(.*?)>/g;

let match;

while (match = reg.exec(str)) {
  // first shows the match: <h1>,h1
  // then shows the match: </h1>,/h1
  alert(match);
}
```

Here we have two matches for `<(.*?)>`, each of them is an array with the full match and groups.

Nested groups

Parentheses can be nested. In this case the numbering also goes from left to right.

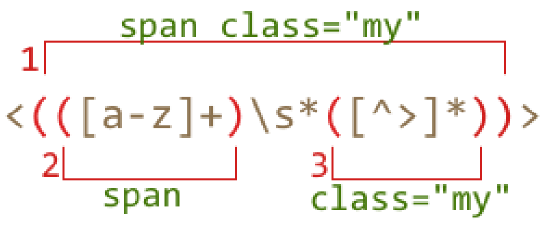
For instance, when searching a tag in `<span class="my">` we may be interested in:

1. The tag content as a whole: `span class="my"`.
2. The tag name: `span`.
3. The tag attributes: `class="my"`.

Let's add parentheses for them:

```
let str = '<span class="my">';

let reg = /<(([a-z]+)\s*([^>]*))>/;

let result = str.match(reg);
alert(result); // <span class="my">, span class="my", span, class="my"
```

Here's how groups look:



At the zero index of the `result` is always the full match.

Then groups, numbered from left to right. Whichever opens first gives the first group `result[1]`. Here it encloses the whole tag content.

Then in `result[2]` goes the group from the second opening `(` till the corresponding `)` – tag name, then we don't group spaces, but group attributes for `result[3]`.

If a group is optional and doesn't exist in the match, the corresponding `result` index is present (and equals `undefined`).

For instance, let's consider the regexp `a(z)?(c)?`. It looks for `"a"` optionally followed by `"z"` optionally followed by `"c"`.

If we run it on the string with a single letter `a`, then the result is:

```
let match = 'a'.match(/a(z)?(c)?/);

alert( match.length ); // 3
alert( match[0] ); // a (whole match)
alert( match[1] ); // undefined
alert( match[2] ); // undefined
```

The array has the length of `3`, but all groups are empty.

And here’s a more complex match for the string `ack`:

```
let match = 'ack'.match(/a(z)?(c)?/)

alert( match.length ); // 3
alert( match[0] ); // ac (whole match)
alert( match[1] ); // undefined, because there's nothing for (z)?
alert( match[2] ); // c
```

The array length is permanent: `3`. But there’s nothing for the group `(z)?`, so the result is `["ac", undefined, "c"]`.

Non-capturing groups with ?:

Sometimes we need parentheses to correctly apply a quantifier, but we don’t want their contents in the array.

A group may be excluded by adding `?:` in the beginning.

For instance, if we want to find `(go)+`, but don’t want to put remember the contents (`go`) in a separate array item, we can write: `(?:go)+`.

In the example below we only get the name “John” as a separate member of the `results` array:

```
let str = "Gogo John!";
// exclude Gogo from capturing
let reg = /(?:go)+ (\w+)/i;

let result = str.match(reg);

alert( result.length ); // 2
alert( result[1] ); // John
```

Tasks

Find color in the format #abc or #abcdef

Write a RegExp that matches colors in the format `#abc` or `#abcdef`. That is: `#` followed by 3 or 6 hexadecimal digits.

Usage example:

```
let reg = /your regexp/g;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(reg) ); // #3f3 #AA00ef
```

P.S. This should be exactly 3 or 6 hex digits: values like `#abcd` should not match.

To solution

Find positive numbers

Create a regexp that looks for positive numbers, including those without a decimal point.

An example of use:

```
let reg = /your regexp/g;

let str = "1.5 0 12. 123.4.";

alert( str.match(reg) );   // 1.5, 0, 12, 123.4
```

To solution

Find all numbers

Write a regexp that looks for all decimal numbers including integer ones, with the floating point and negative ones.

An example of use:

```
let reg = /your regexp/g;

let str = "-1.5 0 2 -123.4.";

alert( str.match(re) ); // -1.5, 0, 2, -123.4
```

To solution

Parse an expression

An arithmetical expression consists of 2 numbers and an operator between them, for instance:

- `1 + 2`
- `1.2 * 3.4`
- `-3 / -6`
- `-2 - 2`

The operator is one of: `"+"`, `"-"`, `"*"` or `"/"`.

There may be extra spaces at the beginning, at the end or between the parts.

Create a function `parse(expr)` that takes an expression and returns an array of 3 items:

1. The first number.
2. The operator.
3. The second number.

For example:

```
let [a, op, b] = parse("1.2 * 3.4");

alert(a); // 1.2
alert(op); // *
alert(b); // 3.4
```

To solution

Backreferences: \n and $n

Capturing groups may be accessed not only in the result, but in the replacement string, and in the pattern too.

Group in replacement: $n

When we are using `replace` method, we can access n-th group in the replacement string using `$n`.

For instance:

```
let name = "John Smith";

name = name.replace(/(\w+) (\w+)/i, "$2, $1");
alert( name ); // Smith, John
```

Here `$1` in the replacement string means “substitute the content of the first group here”, and `$2` means “substitute the second group here”.

Referencing a group in the replacement string allows us to reuse the existing text during the replacement.

Group in pattern: \n

A group can be referenced in the pattern using `\n`.

To make things clear let’s consider a task. We need to find a quoted string: either a single-quoted `'...'` or a double-quoted `"..."` – both variants need to match.

How to look for them?

We can put two kinds of quotes in the pattern: `['"](.*?)['"]`. That finds strings like `"..."` and `'...'`, but it gives incorrect matches when one quote appears inside another one, like the string `"She's the one!"`:

```
let str = "He said: \"She's the one!\".";

let reg = /['"](.*?)['"]/g;

// The result is not what we expect
alert( str.match(reg) ); // "She'
```

As we can see, the pattern found an opening quote `"`, then the text is consumed lazily till the other quote `'`, that closes the match.

To make sure that the pattern looks for the closing quote exactly the same as the opening one, let’s make a group of it and use the backreference:

```
let str = "He said: \"She's the one!\".";

let reg = /(['"])(.*?)\1/g;

alert( str.match(reg) ); // "She's the one!"
```

Now everything’s correct! The regular expression engine finds the first quote `(['"])` and remembers the content of `(...)`, that’s the first capturing group.

Further in the pattern `\1` means “find the same text as in the first group”.

Please note:

- To reference a group inside a replacement string – we use `$1`, while in the pattern – a backslash `\1`.
- If we use `?:` in the group, then we can’t reference it. Groups that are excluded from capturing `(?:...)` are not remembered by the engine.

Alternation (OR) |

Alternation is the term in regular expression that is actually a simple “OR”.

In a regular expression it is denoted with a vertical line character `|`.

For instance, we need to find programming languages: HTML, PHP, Java or JavaScript.

The corresponding regexp: `html|php|java(script)?`.

A usage example:

```
let reg = /html|php|css|java(script)?/gi;

let str = "First HTML appeared, then CSS, then JavaScript";

alert( str.match(reg) ); // 'HTML', 'CSS', 'JavaScript'
```

We already know a similar thing – square brackets. They allow to choose between multiple character, for instance `gr[ae]y` matches `gray` or `grey`.

Alternation works not on a character level, but on expression level. 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 separate a part of the pattern for alternation we usually enclose it in parentheses, like this: `before(XXX|YYY)after`.

Regexp for time

In previous chapters there was a task to build a regexp for searching time in the form `hh:mm`, for instance `12:00`. But a simple `\d\d:\d\d` is too vague. It accepts `25:99` as the time.

How can we make a better one?

We can apply more careful matching:

- The first digit must be `0` or `1` followed by any digit.
- Or `2` followed by `[0-3]`

As a regexp: `[01]\d|2[0-3]`.

Then we can add a colon and the minutes part.

The minutes must be from `0` to `59`, in the regexp language that means the first digit `[0-5]` followed by any other digit `\d`.

Let's glue them together into the pattern: `[01]\d|2[0-3]:[0-5]\d`.

We're almost done, but there's a problem. The alternation `|` is between the `[01]\d` and `2[0-3]:[0-5]\d`. That's wrong, because it will match either the left or the right pattern:

```
let reg = /[01]\d|2[0-3]:[0-5]\d/g;

alert("12".match(reg)); // 12 (matched [01]\d)
```

That's rather obvious, but still an often mistake when starting to work with regular expressions.

We need to add parentheses to apply alternation exactly to hours: `[01]\d` OR `2[0-3]`.

The correct variant:

```
let reg = /([01]\d|2[0-3]):[0-5]\d/g;

alert("00:00 10:10 23:59 25:99 1:2".match(reg)); // 00:00,10:10,23:59
```

Tasks

Find programming languages

There are many programming languages, for instance Java, JavaScript, PHP, C, C++.

Create a regexp that finds them in the string `Java JavaScript PHP C++ C`:

```
let reg = /your regexp/g;

alert("Java JavaScript PHP C++ C".match(reg)); // Java JavaScript PHP C++ C
```

To solution

Find bbtag pairs

A “bb-tag” looks like `[tag]...[/tag]`, where `tag` is one of: `b`, `url` or `quote`.

For instance:

```
[b]text[/b]
[url]http://google.com[/url]
```

BB-tags can be nested. But a tag can’t be nested into itself, for instance:

```
Normal:
[url] [b]http://google.com[/b] [/url]
[quote] [b]text[/b] [/quote]

Impossible:
[b][b]text[/b][/b]
```

Tags can contain line breaks, that’s normal:

```
[quote]
  [b]text[/b]
[/quote]
```

Create a regexp to find all BB-tags with their contents.

For instance:

```
let reg = /your regexp/g;

let str = "..[url]http://google.com[/url]..";
alert( str.match(reg) ); // [url]http://google.com[/url]
```

If tags are nested, then we need the outer tag (if we want we can continue the search in its content):

```
let reg = /your regexp/g;

let str = "..[url][b]http://google.com[/b][/url]..";
alert( str.match(reg) ); // [url][b]http://google.com[/b][/url]
```

To solution

Find quoted strings

Create a regexp to find strings in double quotes `"..."`.

The important part is that strings should support escaping, in the same way as JavaScript strings do. For instance, quotes can be inserted as `\"` a newline as `\n`, and the slash itself as `\\`.

```
let str = "Just like \"here\".";
```

For us it's important that an escaped quote `\"` does not end a string.

So we should look from one quote to the other ignoring escaped quotes on the way.

That's the essential part of the task, otherwise it would be trivial.

Examples of strings to match:

```
.. "test me" ..
.. "Say \"Hello\"!" ... (escaped quotes inside)
.. "\\" ..  (double slash inside)
.. "\\ \"" ..  (double slash and an escaped quote inside)
```

In JavaScript we need to double the slashes to pass them right into the string, like this:

```
let str = ' .. "test me" .. "Say \\"Hello\\"!" .. "\\\\ \\"" .. ';

// the in-memory string
alert(str); //  .. "test me" .. "Say \"Hello\"!" .. "\\ \"" ..
```

To solution

Find the full tag

Write a regexp to find the tag `<style...>`. It should match the full tag: it may have no attributes `<style>` or have several of them `<style type="..." id="...">`.

…But the regexp should not match `<styler>`!

For instance:

```
let reg = /your regexp/g;

alert( '<style> <styler> <style test="...">'.match(reg) ); // <style>, <style test="..
```

To solution

String start ^ and finish $

The caret `'^'` and dollar `'$'` characters have special meaning in a regexp. They are called “anchors”.

The caret `^` matches at the beginning of the text, and the dollar `$` – in the end.

For instance, let’s test if the text starts with `Mary`:

```
let str1 = "Mary had a little lamb, it's fleece was white as snow";
let str2 = 'Everywhere Mary went, the lamp was sure to go';

alert( /^Mary/.test(str1) ); // true
alert( /^Mary/.test(str2) ); // false
```

The pattern `^Mary` means: “the string start and then Mary”.

Now let’s test whether the text ends with an email.

To match an email, we can use a regexp `[-.\w]+@([\w-]+\.)+[\w-]{2,20}`. It’s not perfect, but mostly works.

To test whether the string ends with the email, let’s add `$` to the pattern:

```
let reg = /[-.\w]+@([\w-]+\.)+[\w-]{2,20}$/g;

let str1 = 'My email is mail@site.com';
let str2 = 'Everywhere Mary went, the lamp was sure to go';

alert( reg.test(str1) ); // true
alert( reg.test(str2) ); // false
```

We can use both anchors together to check whether the string exactly follows the pattern. That’s often used for validation.

For instance we want to check that `str` is exactly a color in the form `#` plus 6 hex digits. The pattern for the color is `#[0-9a-f]{6}`.

To check that the *whole string* exactly matches it, we add `^...$`:

```
let str = "#abcdef";

alert( /^#[0-9a-f]{6}$/i.test(str) ); // true
```

The regexp engine looks for the text start, then the color, and then immediately the text end. Just what we need.

Anchors have zero length

Anchors just like `\b` are tests. They have zero-width.

In other words, they do not match a character, but rather force the regexp engine to check the condition (text start/end).

The behavior of anchors changes if there’s a flag `m` (multiline mode). We’ll explore it in the next chapter.

Tasks

Regexp ^$

Which string matches the pattern `^$`?

To solution

Check MAC-address

MAC-address of a network interface consists of 6 two-digit hex numbers separated by a colon.

For instance: `'01:32:54:67:89:AB'`.

Write a regexp that checks whether a string is MAC-address.

Usage:

```
let reg = /your regexp/;

alert( reg.test('01:32:54:67:89:AB') ); // true

alert( reg.test('0132546789AB') ); // false (no colons)

alert( reg.test('01:32:54:67:89') ); // false (5 numbers, must be 6)

alert( reg.test('01:32:54:67:89:ZZ') ) // false (ZZ ad the end)
```

To solution

Multiline mode, flag "m"

The multiline mode is enabled by the flag `/.../m`.

It only affects the behavior of `^` and `$`.

In the multiline mode they match not only at the beginning and end of the string, but also at start/end of line.

Line start ^

In the example below the text has multiple lines. The pattern `/^\d+/gm` takes a number from the beginning of each one:

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert( str.match(/^\d+/gm) ); // 1, 2, 33
```

Without the flag `/.../m` only the first number is matched:

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert( str.match(/^\d+/g) ); // 1
```

That's because by default a caret `^` only matches at the beginning of the text, and in the multiline mode – at the start of a line.

The regular expression engine moves along the text and looks for a string start `^`, when finds – continues to match the rest of the pattern `\d+`.

Line end $

The dollar sign `$` behaves similarly.

The regular expression `\w+$` finds the last word in every line

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert( str.match(/\w+$/gim) ); // Winnie,Piglet,Eeyore
```

Without the `/.../m` flag the dollar `$` would only match the end of the whole string, so only the very last word would be found.

Anchors ^$ versus \n

To find a newline, we can use not only `^` and `$`, but also the newline character `\n`.

The first difference is that unlike anchors, the character `\n` "consumes" the newline character and adds it to the result.

For instance, here we use it instead of `$`:

```
let str = `1st place: Winnie
2nd place: Piglet
33rd place: Eeyore`;

alert( str.match(/\w+\n/gim) ); // Winnie\n,Piglet\n
```

Here every match is a word plus a newline character.

And one more difference – the newline `\n` does not match at the string end. That's why `Eeyore` is not found in the example above.

So, anchors are usually better, they are closer to what we want to get.

Lookahead (in progress)

The article is under development, will be here when it's ready.

Infinite backtracking problem

Some regular expressions are looking simple, but can execute veeeeeery long time, and even "hang" the JavaScript engine.

Sooner or later most developers occasionally face such behavior.

The typical situation – a regular expression works fine sometimes, but for certain strings it "hangs" consuming 100% of CPU.

That may even be a vulnerability. For instance, if JavaScript is on the server, and it uses regular expressions to process user data, then such an input may cause denial of service. The author personally saw and reported such vulnerabilities even for well-known and widely used programs.

So the problem is definitely worth to deal with.

Example

The plan will be like this:

1. First we see the problem how it may occur.
2. Then we simplify the situation and see why it occurs.
3. Then we fix it.

For instance let's consider searching tags in HTML.

We want to find all tags, with or without attributes – like `<a href="..." class="doc" ...>`. We need the regexp to work reliably, because HTML comes from the internet and can be messy.

In particular, we need it to match tags like `<a test="<>" href="#">` – with `<` and `>` in attributes. That's allowed by HTML standard .

Now we can see that a simple regexp like `<[^>]+>` doesn't work, because it stops at the first `>`, and we need to ignore `<>` inside an attribute.

```
// the match doesn't reach the end of the tag - wrong!
alert( '<a test="<>" href="#">'.match(/<[^>]+>/) ); // <a test="<>
```

We need the whole tag.

To correctly handle such situations we need a more complex regular expression. It will have the form `<tag (key=value)*>`.

In the regexp language that is: `<\w+(\s*\w+=(\w+|"[^"]*")\s*)*>`:

1. `<\w+` – is the tag start,
2. `(\s*\w+=(\w+|"[^"]*")\s*)*` – is an arbitrary number of pairs `word=value`, where the value can be either a word `\w+` or a quoted string `"[^"]*"`.

That doesn't yet support few details of HTML grammar, for instance strings in 'single' quotes, but they can be added later, so that's somewhat close to real life. For now we want the regexp to be simple.

Let's try it in action:

```
let reg = /<\w+(\s*\w+=(\w+|"[^"]*")\s*)*>/g;

let str='...<a test="<>" href="#">... <b>...';

alert( str.match(reg) ); // <a test="<>" href="#">, <b>
```

Great, it works! It found both the long tag `<a test="<>" href="#">` and the short one `<b>`.

Now let's see the problem.

If you run the example below, it may hang the browser (or whatever JavaScript engine runs):

```
let reg = /<\w+(\s*\w+=(\w+|"[^"]*")\s*)*>/g;

let str = `<tag a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b
  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b`;

// The search will take a long long time
alert( str.match(reg) );
```

Some regexp engines can handle that search, but most of them don't.

What's the matter? Why a simple regular expression on such a small string "hangs"?

Let's simplify the situation by removing the tag and quoted strings.

Here we look only for attributes:

```
// only search for space-delimited attributes
let reg = /<(\s*\w+=\w+\s*)*>/g;

let str = `<a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b
  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b`;

// the search will take a long, long time
alert( str.match(reg) );
```

The same problem persists.

Here we end the demo of the problem and start looking into what's going on and why it hangs.

Backtracking

To make an example even simpler, let's consider `(\d+)*$`.

This regular expression also has the same problem. In most regexp engines that search takes a very long time (careful – can hang):

```
alert( '12345678901234567890123456789123456789z'.match(/(\d+)*$/) );
```

So what's wrong with the regexp?

First, one may notice that the regexp is a little bit strange. The quantifier `*` looks extraneous. If we want a number, we can use `\d+$`.

Indeed, the regexp is artificial. But the reason why it is slow is the same as those we saw above. So let's understand it, and then return to the real-life examples.

What happen during the search of `(\d+)*$` in the line `123456789z`?

1. First, the regexp engine tries to find a number `\d+`. The plus `+` is greedy by default, so it consumes all digits:

   ```
   \d+.......
   (123456789)z
   ```

2. Then it tries to apply the star around the parentheses `(\d+)*`, but there are no more digits, so it the star doesn't give anything.

   Then the pattern has the string end anchor `$`, and in the text we have `z`.

   ```
              X
   \d+........$
   (123456789)z
   ```

   No match!

3. There's no match, so the greedy quantifier `+` decreases the count of repetitions (backtracks).

   Now `\d+` is not all digits, but all except the last one:

   ```
   \d+.......
   (12345678)9z
   ```

4. Now the engine tries to continue the search from the new position (`9`).

   The start `(\d+)*` can now be applied – it gives the number `9`:

   ```
   \d+.......\d+
   (12345678)(9)z
   ```

   The engine tries to match `$` again, but fails, because meets `z`:

```
             X
\d+.......\d+
(12345678)(9)z
```

There’s no match, so the engine will continue backtracking.

5. Now the first number `\d+` will have 7 digits, and the rest of the string `89` becomes the second `\d+`:

   ```
                X
   \d+......\d+
   (1234567)(89)z
   ```

   …Still no match for `$`.

   The search engine backtracks again. Backtracking generally works like this: the last greedy quantifier decreases the number of repetitions until it can. Then the previous greedy quantifier decreases, and so on. In our case the last greedy quantifier is the second `\d+`, from `89` to `8`, and then the star takes `9`:

   ```
                  X
   \d+......\d+\d+
   (1234567)(8)(9)z
   ```

6. …Fail again. The second and third `\d+` backtracked to the end, so the first quantifier shortens the match to `123456`, and the star takes the rest:

   ```
                X
   \d+.......\d+
   (123456)(789)z
   ```

   Again no match. The process repeats: the last greedy quantifier releases one character (`9`):

   ```
                  X
   \d+.....\d+ \d+
   (123456)(78)(9)z
   ```

7. …And so on.

The regular expression engine goes through all combinations of `123456789` and their subsequences. There are a lot of them, that’s why it takes so long.

A smart guy can say here: “Backtracking? Let’s turn on the lazy mode – and no more backtracking!”.

Let’s replace `\d+` with `\d+?` and see if it works (careful, can hang the browser)

```
// sloooooowwwwww
alert( '123456789012345678901234567891234567892z'.match(/(\d+?)*$/) );
```

No, it doesn’t.

Lazy quantifiers actually do the same, but in the reverse order. Just think about how the search engine would work in this case.

Some regular expression engines have tricky built-in checks to detect infinite backtracking or other means to work around them, but there’s no universal solution.

In the example above, when we search `<(\s*\w+=\w+\s*)*>` in the string `<a=b a=b a=b a=b` – the similar thing happens.

The string has no `>` at the end, so the match is impossible, but the regexp engine does not know about it. The search backtracks trying different combinations of `(\s*\w+=\w+\s*)`:

```
(a=b a=b a=b) (a=b)
(a=b a=b) (a=b a=b)
...
```

How to fix?

The problem – too many variants in backtracking even if we don’t need them.

For instance, in the pattern `(\d+)*$` we (people) can easily see that `(\d+)` does not need to backtrack.

Decreasing the count of `\d+` can not help to find a match, there’s no matter between these two:

```
\d+........
(123456789)z

\d+...\d+....
(1234)(56789)z
```

Let’s get back to more real-life example: `<(\s*\w+=\w+\s*)*>`. We want it to find pairs `name=value` (as many as it can). There’s no need in backtracking here.

In other words, if it found many `name=value` pairs and then can't find `>`, then there's no need to decrease the count of repetitions. Even if we match one pair less, it won't give us the closing `>`:

Modern regexp engines support so-called "possessive" quantifiers for that. They are like greedy, but don't backtrack at all. Pretty simple, they capture whatever they can, and the search continues. There's also another tool called "atomic groups" that forbid backtracking inside parentheses.

Unfortunately, but both these features are not supported by JavaScript.

Although we can get a similar affect using lookahead. 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.

The pattern to take as much repetitions as possible without backtracking is: `(?=(a+))\1`.

In other words, the lookahead `?=` looks for the maximal count `a+` from the current position. And then they are "consumed into the result" by the backreference `\1`.

There will be no backtracking, because lookahead does not backtrack. If it found like 5 times of `a+` and the further match failed, then it doesn't go back to 4.

Let's fix the regexp for a tag with attributes from the beginning of the chapter `<\w+(\s*\w+=(\w+|"[^"]*")\s*)*>`. We'll use lookahead to prevent backtracking of `name=value` pairs:

```
// regexp to search name=value
let attrReg = /(\s*\w+=(\w+|"[^"]*")\s*)/

// use it inside the regexp for tag
let reg = new RegExp('<\\w+(?=(' + attrReg.source + '*))\\1>', 'g');

let good = '...<a test="<>" href="#">... <b>...';

let bad = `<tag a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b
  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b  a=b`;

alert( good.match(reg) ); // <a test="<>" href="#">, <b>
alert( bad.match(reg) ); // null (no results, fast!)
```

Great, it works! We found a long tag `<a test="<>" href="#">` and a small one `<b>` and didn't hang the engine.

Please note the `attrReg.source` property. `RegExp` objects provide access to their source string in it. That's convenient when we want to insert one regexp into another.

Promises, async/await

Introduction: callbacks

Many actions in JavaScript are *asynchronous*.

For instance, take a look at the function `loadScript(src)`:

```
function loadScript(src) {
  let script = document.createElement('script');
  script.src = src;
  document.head.append(script);
}
```

The purpose of the function is to load a new script. When it adds the `<script src="…">` to the document, the browser loads and executes it.

We can use it like this:

```
// loads and executes the script
loadScript('/my/script.js');
```

The function is called “asynchronously,” because the action (script loading) finishes not now, but later.

The call initiates the script loading, then the execution continues. While the script is loading, the code below may finish executing, and if the loading takes time, other scripts may run meanwhile too.

```
loadScript('/my/script.js');
// the code below loadScript doesn't wait for the script loading to finish
// ...
```

Now let’s say we want to use the new script when it loads. It probably declares new functions, so we’d like to run them.

But if we do that immediately after the `loadScript(…)` call, that wouldn’t work:

```
loadScript('/my/script.js'); // the script has "function newFunction() {…}"

newFunction(); // no such function!
```

Naturally, the browser probably didn’t have time to load the script. So the immediate call to the new function fails. As of now, the `loadScript` function doesn’t provide a way to track the load completion. The script loads and eventually runs, that's all. But we'd like to know when it happens, to use new functions and variables from that script.

Let's add a `callback` function as a second argument to `loadScript` that should execute when the script loads:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(script);

  document.head.append(script);
}
```

Now if we want to call new functions from the script, we should write that in the callback:

```
loadScript('/my/script.js', function() {
  // the callback runs after the script is loaded
  newFunction(); // so now it works
  ...
});
```

That's the idea: the second argument is a function (usually anonymous) that runs when the action is completed.

Here's a runnable example with a real script:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;
  script.onload = () => callback(script);
  document.head.append(script);
}

loadScript('https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodash.js', script
  alert(`Cool, the ${script.src} is loaded`);
  alert( _ ); // function declared in the loaded script
});
```

That's called a "callback-based" style of asynchronous programming. A function that does something asynchronously should provide a `callback` argument where we put the function to run after it's complete.

Here we did it in `loadScript`, but of course, it's a general approach.

Callback in callback

How to load two scripts sequentially: the first one, and then the second one after it?

The natural solution would be to put the second `loadScript` call inside the callback, like this:

```
loadScript('/my/script.js', function(script) {

  alert(`Cool, the ${script.src} is loaded, let's load one more`);

  loadScript('/my/script2.js', function(script) {
    alert(`Cool, the second script is loaded`);
  });

});
```

After the outer `loadScript` is complete, the callback initiates the inner one.

What if we want one more script…?

```
loadScript('/my/script.js', function(script) {

  loadScript('/my/script2.js', function(script) {

    loadScript('/my/script3.js', function(script) {
      // ...continue after all scripts are loaded
    });

  })

});
```

So, every new action is inside a callback. That’s fine for few actions, but not good for many, so we’ll see other variants soon.

Handling errors

In examples above we didn’t consider errors. What if the script loading fails? Our callback should be able to react on that.

Here’s an improved version of `loadScript` that tracks loading errors:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error for ${src}`));

  document.head.append(script);
}
```

It calls `callback(null, script)` for successful load and `callback(error)` otherwise.

The usage:

```
loadScript('/my/script.js', function(error, script) {
  if (error) {
    // handle error
  } else {
    // script loaded successfully
  }
});
```

Once again, the recipe that we used for `loadScript` is actually quite common. It's called the "error-first callback" style.

The convention is:

1. The first argument of the `callback` is reserved for an error if it occurs. Then `callback(err)` is called.
2. The second argument (and the next ones if needed) are for the successful result. Then `callback(null, result1, result2…)` is called.

So the single `callback` function is used both for reporting errors and passing back results.

Pyramid of Doom

From the first look, it's a viable way of asynchronous coding. And indeed it is. For one or maybe two nested calls it looks fine.

But for multiple asynchronous actions that follow one after another we'll have code like this:

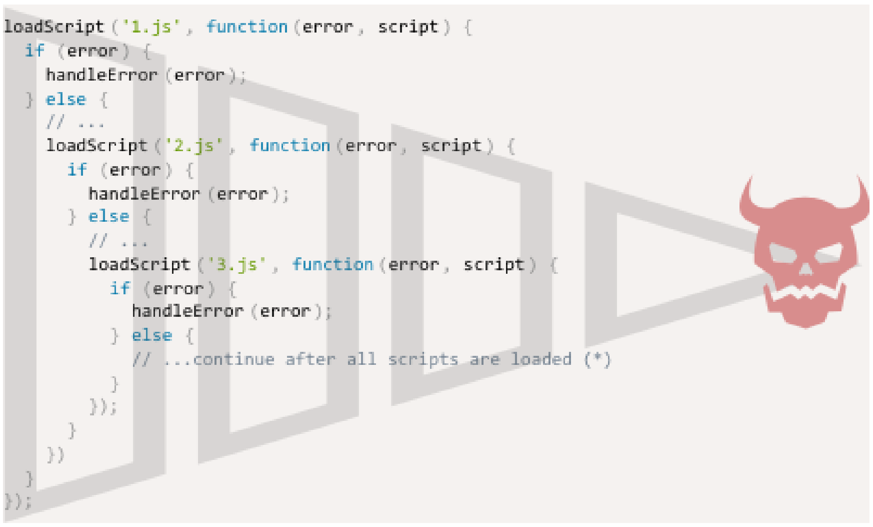
```
loadScript('1.js', function(error, script) {

  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });

      }
    })
  }
});
```

In the code above:

1. We load `1.js`, then if there’s no error.
2. We load `2.js`, then if there’s no error.
3. We load `3.js`, then if there’s no error – do something else `(*)`.

As calls become more nested, the code becomes deeper and increasingly more difficult to manage, especially if we have a real code instead of `...`, that may include more loops, conditional statements and so on.

That’s sometimes called “callback hell” or “pyramid of doom.”



```
loadScript('1.js', function(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', function(error, script) {
      if (error) {
        handleError(error);
      } else {
        // ...
        loadScript('3.js', function(error, script) {
          if (error) {
            handleError(error);
          } else {
            // ...continue after all scripts are loaded (*)
          }
        });
      }
    })
  }
});
```

The “pyramid” of nested calls grows to the right with every asynchronous action. Soon it spirals out of control.

So this way of coding isn’t very good.

We can try to alleviate the problem by making every action a standalone function, like this:

```
loadScript('1.js', step1);

function step1(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('2.js', step2);
  }
}

function step2(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...
    loadScript('3.js', step3);
  }
}

function step3(error, script) {
  if (error) {
    handleError(error);
  } else {
    // ...continue after all scripts are loaded (*)
  }
};
```

See? It does the same, and there's no deep nesting now because we made every action a separate top-level function.

It works, but the code looks like a torn apart spreadsheet. It's difficult to read, and you probably noticed that. One needs to eye-jump between pieces while reading it. That's inconvenient, especially if the reader is not familiar with the code and doesn't know where to eye-jump.

Also, the functions named `step*` are all of single use, they are created only to avoid the "pyramid of doom." No one is going to reuse them outside of the action chain. So there's a bit of a namespace cluttering here.

We'd like to have something better.

Luckily, there are other ways to avoid such pyramids. One of the best ways is to use "promises," described in the next chapter.

Tasks

Animated circle with callback

In 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.

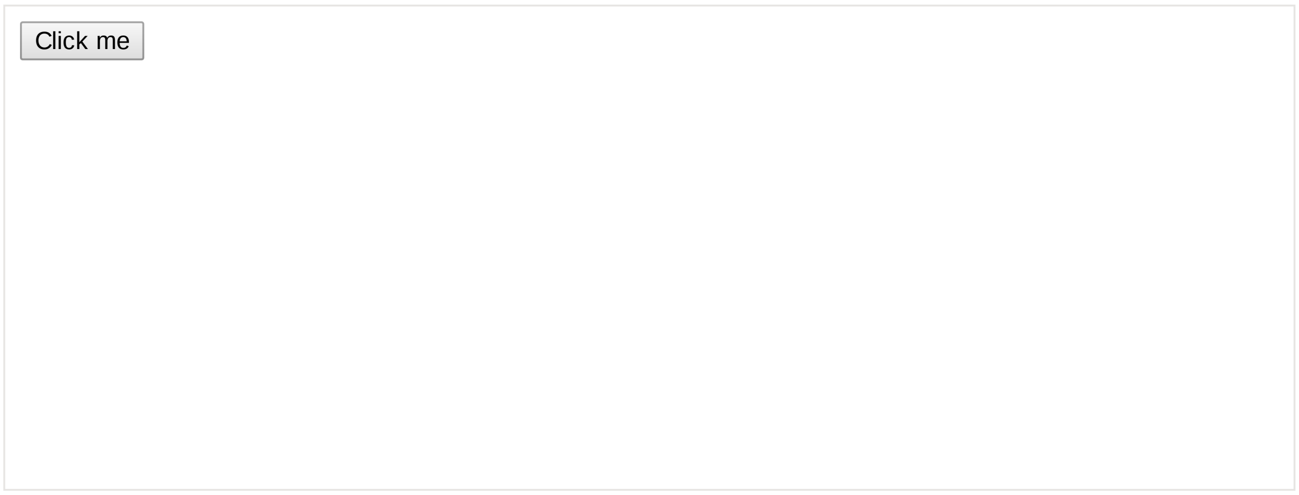
In the solution of the task, the function `showCircle(cx, cy, radius)` draws the circle, but gives no way to track when it’s ready.

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.

To solution

Promise

Imagine that you’re a top singer, and fans ask day and night for your upcoming single.

To get some relief, you promise to send it to them when it’s published. You give your fans a list to which they can subscribe for updates. They can fill in their email addresses, so that when the song becomes available, all subscribed parties instantly receive it. And even if something goes very wrong, say, if plans to publish the song are cancelled, they will still be notified.

Everyone is happy, because the people don't crowd you any more, and fans, because they won't miss the single.

This is a real-life analogy for things we often have in programming:

1. A "producing code" that does something and takes time. For instance, the code loads a remote script. That's a "singer".
2. A "consuming code" that wants the result of the "producing code" once it's ready. Many functions may need that result. These are the "fans".
3. A *promise* is a special JavaScript object that links the "producing code" and the "consuming code" together. In terms of our analogy: this is the "subscription list". The "producing code" takes whatever time it needs to produce the promised result, and the "promise" makes that result available to all of the subscribed code when it's ready.

The analogy isn't terribly accurate, because JavaScript promises are more complex than a simple subscription list: they have additional features and limitations. But it's fine to begin with.

The constructor syntax for a promise object is:

```
let promise = new Promise(function(resolve, reject) {
  // executor (the producing code, "singer")
});
```

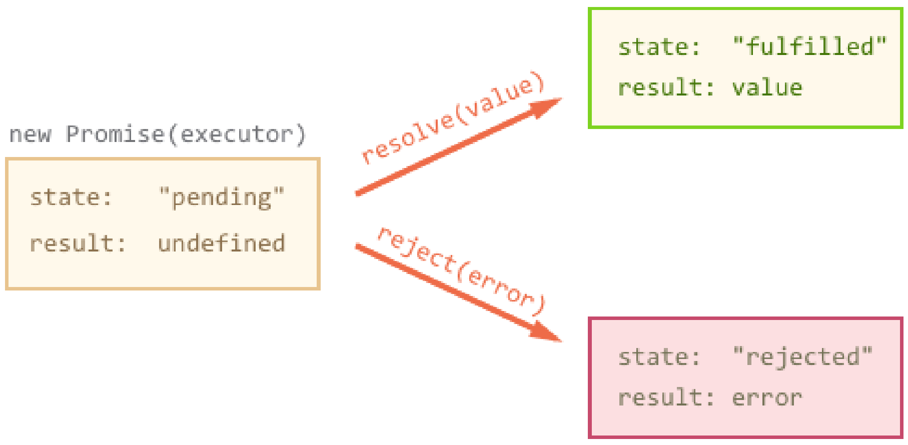
The function passed to `new Promise` is called the *executor*. When the promise is created, this executor function runs automatically. It contains the producing code, that should eventually produce a result. In terms of the analogy above: the executor is the "singer".

The resulting `promise` object has internal properties:

- `state` — initially "pending", then changes to either "fulfilled" or "rejected",
- `result` — an arbitrary value of your choosing, initially `undefined`.

When the executor finishes the job, it should call one of the functions that it gets as arguments:

- `resolve(value)` — to indicate that the job finished successfully:
  - sets `state` to `"fulfilled"`,
  - sets `result` to `value`.
- `reject(error)` — to indicate that an error occurred:
  - sets `state` to `"rejected"`,
  - sets `result` to `error`.



Later we'll see how these changes become known to "fans".

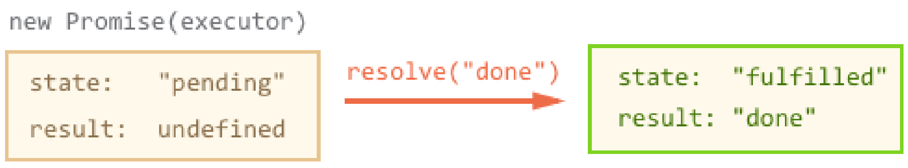
Here's an example of a Promise constructor and a simple executor function with its "producing code" (the `setTimeout`):

```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done!"
  setTimeout(() => resolve("done!"), 1000);
});
```

We can see two things by running the code above:

1. The executor is called automatically and immediately (by the `new Promise`).
2. The executor receives two arguments: `resolve` and `reject` — these functions are pre-defined by the JavaScript engine. So we don't need to create them. Instead, we should write the executor to call them when ready.

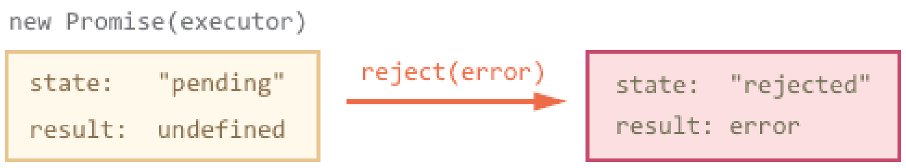
After one second of "processing" the executor calls `resolve("done")` to produce the result:



That was an example of a successful job completion, a "fulfilled promise".

And now an example of the executor rejecting the promise with an error:

```
let promise = new Promise(function(resolve, reject) {
  // after 1 second signal that the job is finished with an error
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```



To summarize, the executor should do a job (something that takes time usually) and then call `resolve` or `reject` to change the state of the corresponding Promise object.

The Promise that is either resolved or rejected is called “settled”, as opposed to a “pending” Promise.

There can be only a single result or an error

The executor should call only one `resolve` or `reject`. The promise’s state change is final.

All further calls of `resolve` and `reject` are ignored:

```
let promise = new Promise(function(resolve, reject) {
  resolve("done");

  reject(new Error("…")); // ignored
  setTimeout(() => resolve("…")); // ignored
});
```

The idea is that a job done by the executor may have only one result or an error.

Further, `resolve`/`reject` expect only one argument and will ignore additional arguments.

Reject with `Error` objects

In case if something goes wrong, we can call `reject` with any type of argument (just like `resolve`). But it is recommended to use `Error` objects (or objects that inherit from `Error`). The reasoning for that will soon become apparent.

Immediately calling `resolve`/`reject`

In practice, an executor usually does something asynchronously and calls `resolve`/`reject` after some time, but it doesn't have to. We also can call `resolve` or `reject` immediately, like this:

```
let promise = new Promise(function(resolve, reject) {
  // not taking our time to do the job
  resolve(123); // immediately give the result: 123
});
```

For instance, this might happen when we start to do a job but then see that everything has already been completed.

That's fine. We immediately have a resolved Promise, nothing wrong with that.

The `state` and `result` are internal

The properties `state` and `result` of the Promise object are internal. We can't directly access them from our "consuming code". We can use the methods `.then`/`.catch` for that. They are described below.

Consumers: "then" and "catch"

A Promise object serves as a link between the executor (the "producing code" or "singer") and the consuming functions (the "fans"), which will receive the result or error. Consuming functions can be registered (subscribed) using the methods `.then` and `.catch`.

The syntax of `.then` is:

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

The first argument of `.then` is a function that:

1. runs when the Promise is resolved, and
2. receives the result.

The second argument of `.then` is a function that:

1. runs when the Promise is rejected, and

2. receives the error.

For instance, here's the reaction to a successfuly resolved promise:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve("done!"), 1000);
});

// resolve runs the first function in .then
promise.then(
  result => alert(result), // shows "done!" after 1 second
  error => alert(error) // doesn't run
);
```

The first function was executed.

And in the case of a rejection – the second one:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

If we're interested only in successful completions, then we can provide only one function argument to `.then`:

```
let promise = new Promise(resolve => {
  setTimeout(() => resolve("done!"), 1000);
});

promise.then(alert); // shows "done!" after 1 second
```

If we're interested only in errors, then we can use `null` as the first argument: `.then(null, errorHandlingFunction)`. Or we can use `.catch(errorHandlingFunction)`, which is exactly the same:

```
let promise = new Promise((resolve, reject) => {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

```

```
// .catch(f) is the same as promise.then(null, f)
promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

The call `.catch(f)` is a complete analog of `.then(null, f)`, it's just a shorthand.

On settled promises `then` runs immediately

If a promise is pending, `.then/catch` handlers wait for the result. Otherwise, if a promise has already settled, they execute immediately:

```
// an immediately resolved promise
let promise = new Promise(resolve => resolve("done!"));

promise.then(alert); // done! (shows up right now)
```

Some tasks may sometimes require time and sometimes finish immediately. The good thing is: the `.then` handler is guaranteed to run in both cases.

Handlers of `.then` / `.catch` are always asynchronous

Even when the Promise is immediately resolved, code which occurs on lines *below* your `.then` / `.catch` may still execute first.

The JavaScript engine has an internal execution queue which gets all `.then/catch` handlers.

But it only looks into that queue when the current execution is finished.

In other words, `.then/catch` handlers are pending execution until the engine is done with the current code.

For instance, here:

```
// an "immediately" resolved Promise
const executor = resolve => resolve("done!");
const promise = new Promise(executor);

promise.then(alert); // this alert shows last (*)

alert("code finished"); // this alert shows first
```

The promise becomes settled immediately, but the engine first finishes the current code, calls `alert`, and only *afterwards* looks into the queue to run `.then` handler.

So the code *after* `.then` ends up always running *before* the Promise's subscribers, even in the case of an immediately-resolved Promise.

Usually that's unimportant, but in some scenarios the order may matter a great deal.

Next, let's see more practical examples of how promises can help us to write asynchronous code.

Example: loadScript

We've got the `loadScript` function for loading a script from the previous chapter.

Here's the callback-based variant, just to remind us of it:

```
function loadScript(src, callback) {
  let script = document.createElement('script');
  script.src = src;

  script.onload = () => callback(null, script);
  script.onerror = () => callback(new Error(`Script load error ` + src));
  document.head.append(script);
}
```

Let’s rewrite it using Promises.

The new function `loadScript` will not require a callback. Instead, it will create and return a Promise object that resolves when the loading is complete. The outer code can add handlers (subscribing functions) to it using `.then`:

```
function loadScript(src) {
  return new Promise(function(resolve, reject) {
    let script = document.createElement('script');
    script.src = src;

    script.onload = () => resolve(script);
    script.onerror = () => reject(new Error("Script load error: " + src));

    document.head.append(script);
  });
}
```

Usage:

```
let promise = loadScript("https://cdnjs.cloudflare.com/ajax/libs/lodash.js/3.2.0/lodas

promise.then(
  script => alert(`${script.src} is loaded!`),
  error => alert(`Error: ${error.message}`)
);

promise.then(script => alert('One more handler to do something else!'));
```

We can immediately see a few benefits over the callback-based pattern:

Demerits	Merits
• We must have a ready `callback` function when calling `loadScript`. In other words, we must know what to do with the result *before* `loadScript` is called. • There can be only one callback.	• Promises allow us to do things in the natural order. First, we run `loadScript`, and `.then` we write what to do with the result. • We can call `.then` on a Promise as many times as we want. Each time, we’re adding a new “fan”, a new subscribing function, to the “subscription list”.

More about this in the next section: Promise Chaining.

So Promises already give us better code flow and flexibility. But there’s more. We’ll see that in the next chapters.

✓ Tasks

Re-resolve a promise?

What’s the output of the code below?

```
let promise = new Promise(function(resolve, reject) {
  resolve(1);

  setTimeout(() => resolve(2), 1000);
});

promise.then(alert);
```

To solution

Delay with a promise

The built-in function `setTimeout` uses callbacks. Create a promise-based alternative.

The function `delay(ms)` should return a promise. That promise should resolve after `ms` milliseconds, so that we can add `.then` to it, like this:

```
function delay(ms) {
  // your code
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

To solution

Animated circle with promise

Rewrite the `showCircle` function in the solution of the task Animated circle with callback so that it returns a promise instead of accepting a callback.

The new usage:

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

Take the solution of the task Animated circle with callback as the base.

To solution

Promises chaining

Let’s return to the problem mentioned in the chapter Introduction: callbacks.

- We have a sequence of asynchronous tasks to be done one after another. For instance, loading scripts.
- How to code it well?

Promises provide a couple of recipes to do that.

In this chapter we cover promise chaining.

It looks like this:

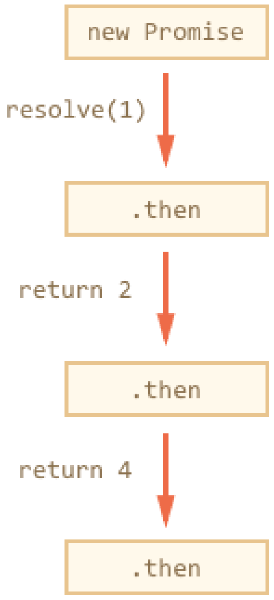
```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000); // (*)

}).then(function(result) { // (**)

  alert(result); // 1
  return result * 2;

}).then(function(result) { // (***)

  alert(result); // 2
  return result * 2;

}).then(function(result) {

  alert(result); // 4
  return result * 2;

});
```

The idea is that the result is passed through the chain of `.then` handlers.

Here the flow is:

1. The initial promise resolves in 1 second `(*)`,
2. Then the `.then` handler is called `(**)`.
3. The value that it returns is passed to the next `.then` handler `(***)`
4. …and so on.

As the result is passed along the chain of handlers, we can see a sequence of `alert` calls: `1` → `2` → `4`.



The whole thing works, because a call to `promise.then` returns a promise, so that we can call the next `.then` on it.

When a handler returns a value, it becomes the result of that promise, so the next `.then` is called with it.

To make these words more clear, here’s the start of the chain:

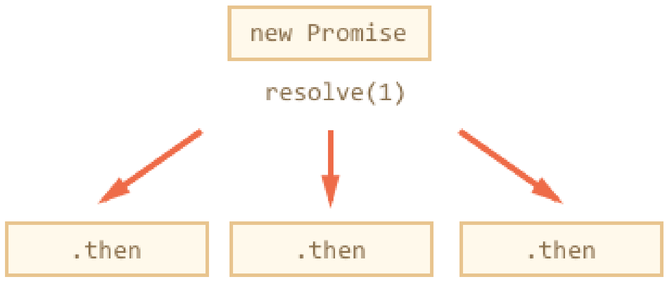
```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result);
  return result * 2; // <-- (1)

}) // <-- (2)
// .then…
```

The value returned by `.then` is a promise, that’s why we are able to add another `.then` at `(2)`. When the value is returned in `(1)`, that promise becomes resolved, so the next handler runs with the value.

Unlike the chaining, technically we can also add many `.then` to a single promise, like this:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => resolve(1), 1000);
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});

promise.then(function(result) {
  alert(result); // 1
  return result * 2;
});
```

…But that's a totally different thing. Here's the picture (compare it with the chaining above):



All `.then` on the same promise get the same result – the result of that promise. So in the code above all `alert` show the same: `1`. There is no result-passing between them.

In practice we rarely need multiple handlers for one promise. Chaining is used much more often.

Returning promises

Normally, a value returned by a `.then` handler is immediately passed to the next handler. But there's an exception.

If the returned value is a promise, then the further execution is suspended until it settles. After that, the result of that promise is given to the next `.then` handler.

For instance:

```
new Promise(function(resolve, reject) {

  setTimeout(() => resolve(1), 1000);

}).then(function(result) {

  alert(result); // 1

  return new Promise((resolve, reject) => { // (*)
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) { // (**)

  alert(result); // 2

  return new Promise((resolve, reject) => {
    setTimeout(() => resolve(result * 2), 1000);
  });

}).then(function(result) {

  alert(result); // 4

});
```

Here the first `.then` shows `1` returns `new Promise(…)` in the line `(*)`. After one second it resolves, and the result (the argument of `resolve`, here it's `result*2`) is passed on to handler of the second `.then` in the line `(**)`. It shows `2` and does the same thing.

So the output is again 1 → 2 → 4, but now with 1 second delay between `alert` calls.

Returning promises allows us to build chains of asynchronous actions.

Example: loadScript

Let's use this feature with `loadScript` to load scripts one by one, in sequence:

```
loadScript("/article/promise-chaining/one.js")
  .then(function(script) {
    return loadScript("/article/promise-chaining/two.js");
  })
  .then(function(script) {
    return loadScript("/article/promise-chaining/three.js");
  })
  .then(function(script) {
    // use functions declared in scripts
    // to show that they indeed loaded
    one();
    two();
    three();
  });
```

Here each `loadScript` call returns a promise, and the next `.then` runs when it resolves. Then it initiates the loading of the next script. So scripts are loaded one after another.

We can add more asynchronous actions to the chain. Please note that code is still “flat”, it grows down, not to the right. There are no signs of “pyramid of doom”.

Please note that technically it is also possible to write `.then` directly after each promise, without returning them, like this:

```
loadScript("/article/promise-chaining/one.js").then(function(script1) {
  loadScript("/article/promise-chaining/two.js").then(function(script2) {
    loadScript("/article/promise-chaining/three.js").then(function(script3) {
      // this function has access to variables script1, script2 and script3
      one();
      two();
      three();
    });
  });
});
```

This code does the same: loads 3 scripts in sequence. But it “grows to the right”. So we have the same problem as with callbacks. Use chaining (return promises from `.then`) to evade it.

Sometimes it’s ok to write `.then` directly, because the nested function has access to the outer scope (here the most nested callback has access to all variables `scriptX`), but that’s an exception rather than a rule.

Thenables

To be precise, `.then` may return an arbitrary “thenable” object, and it will be treated the same way as a promise.

A “thenable” object is any object with a method `.then`.

The idea is that 3rd-party libraries may implement “promise-compatible” objects of their own. They can have extended set of methods, but also be compatible with native promises, because they implement `.then`.

Here’s an example of a thenable object:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // resolve with this.num*2 after the 1 second
    setTimeout(() => resolve(this.num * 2), 1000); // (**)
  }
}

new Promise(resolve => resolve(1))
  .then(result => {
    return new Thenable(result); // (*)
  })
  .then(alert); // shows 2 after 1000ms
```

JavaScript checks the object returned by `.then` handler in the line `(*)`: if it has a callable method named `then`, then it calls that method providing native functions `resolve`, `reject` as arguments (similar to executor) and waits until one of them is called. In the example above `resolve(2)` is called after 1 second `(**)`. Then the result is passed further down the chain.

This feature allows to integrate custom objects with promise chains without having to inherit from `Promise`.

Bigger example: fetch

In frontend programming promises are often used for network requests. So let’s see an extended example of that.

We’ll use the fetch method to load the information about the user from the remote server. The method is quite complex, it has many optional parameters, but the basic usage is quite simple:

```
let promise = fetch(url);
```

This makes a network request to the `url` and returns a promise. The promise resolves with a `response` object when the remote server responds with headers, but *before the full response is downloaded*.

To read the full response, we should call a method `response.text()`: it returns a promise that resolves when the full text downloaded from the remote server, with that text as a result.

The code below makes a request to `user.json` and loads its text from the server:

```
fetch('/article/promise-chaining/user.json')
  // .then below runs when the remote server responds
  .then(function(response) {
    // response.text() returns a new promise that resolves with the full response text
    // when we finish downloading it
    return response.text();
  })
  .then(function(text) {
    // ...and here's the content of the remote file
    alert(text); // {"name": "iliakan", isAdmin: true}
  });
```

There is also a method `response.json()` that reads the remote data and parses it as JSON. In our case that's even more convenient, so let's switch to it.

We'll also use arrow functions for brevity:

```
// same as above, but response.json() parses the remote content as JSON
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => alert(user.name)); // iliakan
```

Now let's do something with the loaded user.

For instance, we can make one more request to github, load the user profile and show the avatar:

```
// Make a request for user.json
fetch('/article/promise-chaining/user.json')
  // Load it as json
  .then(response => response.json())
  // Make a request to github
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  // Load the response as json
  .then(response => response.json())
  // Show the avatar image (githubUser.avatar_url) for 3 seconds (maybe animate it)
  .then(githubUser => {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => img.remove(), 3000); // (*)
  });
```

The code works, see comments about the details, but it should be quite self-descriptive. Although, there’s a potential problem in it, a typical error of those who begin to use promises.

Look at the line `(*)`: how can we do something *after* the avatar has finished showing and gets removed? For instance, we’d like to show a form for editing that user or something else. As of now, there’s no way.

To make the chain extendable, we need to return a promise that resolves when the avatar finishes showing.

Like this:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  // triggers after 3 seconds
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
```

Now right after `setTimeout` runs `img.remove()`, it calls `resolve(githubUser)`, thus passing the control to the next `.then` in the chain and passing forward the user data.

As a rule, an asynchronous action should always return a promise.

That makes possible to plan actions after it. Even if we don’t plan to extend the chain now, we may need it later.

Finally, we can split the code into reusable functions:

```
function loadJson(url) {
  return fetch(url)
    .then(response => response.json());
}

function loadGithubUser(name) {
  return fetch(`https://api.github.com/users/${name}`)
    .then(response => response.json());
}

function showAvatar(githubUser) {
  return new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  });
}

// Use them:
loadJson('/article/promise-chaining/user.json')
  .then(user => loadGithubUser(user.name))
  .then(showAvatar)
  .then(githubUser => alert(`Finished showing ${githubUser.name}`));
  // ...
```

Error handling

Asynchronous actions may sometimes fail: in case of an error the corresponding promise becomes rejected. For instance, `fetch` fails if the remote server is not available. We can use `.catch` to handle errors (rejections).

Promise chaining is great at that aspect. When a promise rejects, the control jumps to the closest rejection handler down the chain. That’s very convenient in practice.

For instance, in the code below the URL is wrong (no such server) and `.catch` handles the error:

```
fetch('https://no-such-server.blabla') // rejects
  .then(response => response.json())
  .catch(err => alert(err)) // TypeError: failed to fetch (the text may vary)
```

Or, maybe, everything is all right with the server, but the response is not a valid JSON:

```
fetch('/') // fetch works fine now, the server responds successfully
  .then(response => response.json()) // rejects: the page is HTML, not a valid json
  .catch(err => alert(err)) // SyntaxError: Unexpected token < in JSON at position 0
```

In the example below we append `.catch` to handle all errors in the avatar-loading-and-showing chain:

```
fetch('/article/promise-chaining/user.json')
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`))
  .then(response => response.json())
  .then(githubUser => new Promise(function(resolve, reject) {
    let img = document.createElement('img');
    img.src = githubUser.avatar_url;
    img.className = "promise-avatar-example";
    document.body.append(img);

    setTimeout(() => {
      img.remove();
      resolve(githubUser);
    }, 3000);
  }))
  .catch(error => alert(error.message));
```

Here `.catch` doesn't trigger at all, because there are no errors. But if any of the promises above rejects, then it would execute.

Implicit try…catch

The code of the executor and promise handlers has an "invisible `try..catch`" around it. If an error happens, it gets caught and treated as a rejection.

For instance, this code:

```
new Promise(function(resolve, reject) {
  throw new Error("Whoops!");
}).catch(alert); // Error: Whoops!
```

…Works the same way as this:

```
new Promise(function(resolve, reject) {
  reject(new Error("Whoops!"));
}).catch(alert); // Error: Whoops!
```

The "invisible `try..catch`" around the executor automatically catches the error and treats it as a rejection.

That's so not only in the executor, but in handlers as well. If we `throw` inside `.then` handler, that means a rejected promise, so the control jumps to the nearest error handler.

Here's an example:

```
new Promise(function(resolve, reject) {
  resolve("ok");
}).then(function(result) {
  throw new Error("Whoops!"); // rejects the promise
}).catch(alert); // Error: Whoops!
```

That's so not only for `throw`, but for any errors, including programming errors as well:

```
new Promise(function(resolve, reject) {
  resolve("ok");
}).then(function(result) {
  blabla(); // no such function
}).catch(alert); // ReferenceError: blabla is not defined
```

As a side effect, the final `.catch` not only catches explicit rejections, but also occasional errors in the handlers above.

Rethrowing

As we already noticed, `.catch` behaves like `try..catch`. We may have as many `.then` as we want, and then use a single `.catch` at the end to handle errors in all of them.

In a regular `try..catch` we can analyze the error and maybe rethrow it if can't handle. The same thing is possible for promises. If we `throw` inside `.catch`, then the control goes to the next closest error handler. And if we handle the error and finish normally, then it continues to the closest successful `.then` handler.

In the example below the `.catch` successfully handles the error:

```
// the execution: catch -> then
new Promise(function(resolve, reject) {

  throw new Error("Whoops!");

}).catch(function(error) {

  alert("The error is handled, continue normally");
}).then(() => alert("Next successful handler runs"));
```

Here the `.catch` block finishes normally. So the next successful handler is called. Or it could return something, that would be the same.

…And here the `.catch` block analyzes the error and throws it again:

```
// the execution: catch -> catch -> then
new Promise(function(resolve, reject) {

  throw new Error("Whoops!");

}).catch(function(error) { // (*)

  if (error instanceof URIError) {
    // handle it
  } else {
    alert("Can't handle such error");

    throw error; // throwing this or another error jumps to the next catch
  }

}).then(function() {
  /* never runs here */
}).catch(error => { // (**)

  alert(`The unknown error has occurred: ${error}`);
  // don't return anything => execution goes the normal way

});
```

The handler `(*)` catches the error and just can't handle it, because it's not `URIError`, so it throws it again. Then the execution jumps to the next `.catch` down the chain `(**)`.

In the section below we'll see a practical example of rethrowing.

Fetch error handling example

Let's improve error handling for the user-loading example.

The promise returned by fetch rejects when it's impossible to make a request. For instance, a remote server is not available, or the URL is malformed. But if the remote server responds with error 404, or even error 500, then it's considered a valid response.

What if the server returns a non-JSON page with error 500 in the line `(*)`? What if there's no such user, and github returns a page with error 404 at `(**)`?

```
fetch('no-such-user.json') // (*)
  .then(response => response.json())
  .then(user => fetch(`https://api.github.com/users/${user.name}`)) // (**)
  .then(response => response.json())
  .catch(alert); // SyntaxError: Unexpected token < in JSON at position 0
  // ...
```

As of now, the code tries to load the response as JSON no matter what and dies with a syntax error. You can see that by running the example above, as the file `no-such-user.json` doesn't exist.

That's not good, because the error just falls through the chain, without details: what failed and where.

So let's add one more step: we should check the `response.status` property that has HTTP status, and if it's not 200, then throw an error.

```
class HttpError extends Error { // (1)
  constructor(response) {
    super(`${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) { // (2)
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    })
}

loadJson('no-such-user.json') // (3)
  .catch(alert); // HttpError: 404 for .../no-such-user.json
```

1. We make a custom class for HTTP Errors to distinguish them from other types of errors. Besides, the new class has a constructor that accepts the `response` object and saves it in the error. So error-handling code will be able to access it.
2. Then we put together the requesting and error-handling code into a function that fetches the `url` *and* treats any non-200 status as an error. That's convenient, because we often need such logic.
3. Now `alert` shows better message.

The great thing about having our own class for errors is that we can easily check for it in error-handling code.

For instance, we can make a request, and then if we get 404 – ask the user to modify the information.

The code below loads a user with the given name from github. If there's no such user, then it asks for the correct name:

```
function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Full name: ${user.name}.`); // (1)
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) { // (2)
        alert("No such user, please reenter.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

Here:

1. If `loadJson` returns a valid user object, then the name is shown `(1)`, and the user is returned, so that we can add more user-related actions to the chain. In that case the `.catch` below is ignored, everything's very simple and fine.
2. Otherwise, in case of an error, we check it in the line `(2)`. Only if it's indeed the HTTP error, and the status is 404 (Not found), we ask the user to reenter. For other errors – we don't know how to handle, so we just rethrow them.

Unhandled rejections

What happens when an error is not handled? For instance, after the rethrow as in the example above. Or if we forget to append an error handler to the end of the chain, like here:

```
new Promise(function() {
  noSuchFunction(); // Error here (no such function)
}); // no .catch attached
```

Or here:

```
// a chain of promises without .catch at the end
new Promise(function() {
  throw new Error("Whoops!");
}).then(function() {
  // ...something...
}).then(function() {
  // ...something else...
}).then(function() {
  // ...but no catch after it!
});
```

In case of an error, the promise state becomes “rejected”, and the execution should jump to the closest rejection handler. But there is no such handler in the examples above. So the error gets “stuck”.

In practice, that’s usually because of the bad code. Indeed, how come there’s no error handling?

Most JavaScript engines track such situations and generate a global error in that case. We can see it in the console.

In the browser we can catch it using the event `unhandledrejection`:

```
window.addEventListener('unhandledrejection', function(event) {
  // the event object has two special properties:
  alert(event.promise); // [object Promise] - the promise that generated the error
  alert(event.reason); // Error: Whoops! - the unhandled error object
});

new Promise(function() {
  throw new Error("Whoops!");
}); // no catch to handle the error
```

The event is the part of the HTML standard . Now if an error occurs, and there’s no `.catch`, the `unhandledrejection` handler triggers: the `event` object has the information about the error, so we can do something with it.

Usually such errors are unrecoverable, so our best way out is to inform the user about the problem and probably report about the incident to the server.

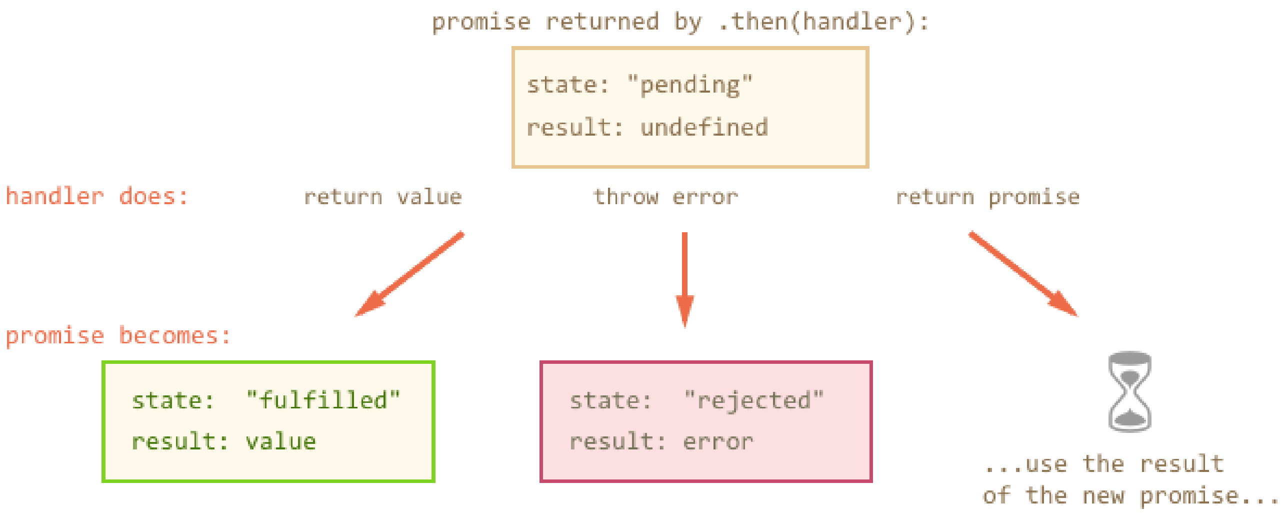
In non-browser environments like Node.JS there are other similar ways to track unhandled errors.

Summary

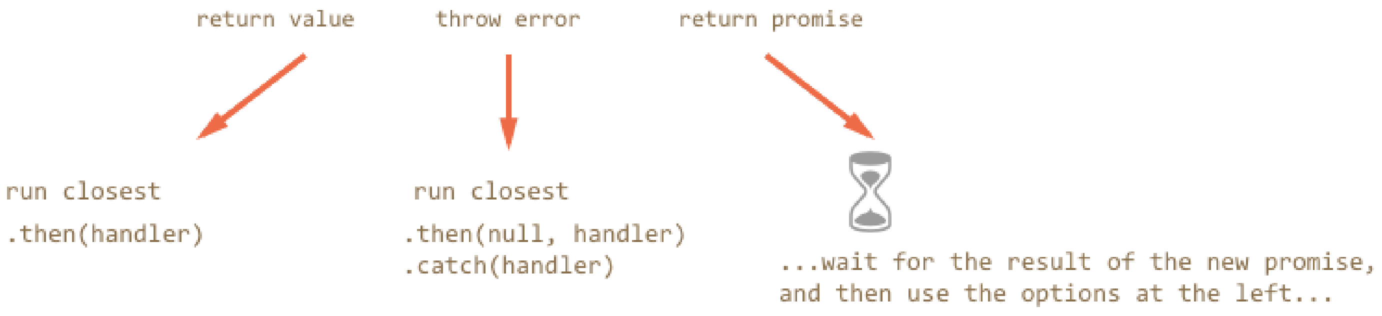
To summarize, `.then/catch(handler)` returns a new promise that changes depending on what handler does:

1. If it returns a value or finishes without a `return` (same as `return undefined`), then the new promise becomes resolved, and the closest resolve handler (the first argument of `.then`) is called with that value.
2. If it throws an error, then the new promise becomes rejected, and the closest rejection handler (second argument of `.then` or `.catch`) is called with it.
3. If it returns a promise, then JavaScript waits until it settles and then acts on its outcome the same way.

The picture of how the promise returned by `.then/catch` changes:



The smaller picture of how handlers are called:



In the examples of error handling above the `.catch` was always the last in the chain. In practice though, not every promise chain has a `.catch`. Just like regular code is not always wrapped in `try..catch`.

We should place `.catch` exactly in the places where we want to handle errors and know how to handle them. Using custom error classes can help to analyze errors and rethrow those that we can’t handle.

For errors that fall outside of our scope we should have the `unhandledrejection` event handler (for browsers, and analogs for other environments). Such unknown errors are usually unrecoverable, so all we should do is to inform the user and probably report to our server about the incident.

✓ Tasks

Promise: then versus catch

Are these code fragments equal? In other words, do they behave the same way in any circumstances, for any handler functions?

```
promise.then(f1, f2);
```

Versus;

```
promise.then(f1).catch(f2);
```

To solution

Error in setTimeout

What do you think? Will the `.catch` trigger? Explain your answer.

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

To solution

Promise API

There are 4 static methods in the `Promise` class. We’ll quickly cover their use cases here.

Promise.resolve

The syntax:

```
let promise = Promise.resolve(value);
```

Returns a resolved promise with the given `value`.

Same as:

```
let promise = new Promise(resolve => resolve(value));
```

The method is used when we already have a value, but would like to have it “wrapped” into a promise.

For instance, the `loadCached` function below fetches the `url` and remembers the result, so that future calls on the same URL return it immediately:

```
function loadCached(url) {
  let cache = loadCached.cache || (loadCached.cache = new Map());

  if (cache.has(url)) {
    return Promise.resolve(cache.get(url)); // (*)
  }

  return fetch(url)
    .then(response => response.text())
    .then(text => {
      cache.set(url,text);
      return text;
    });
}
```

We can use `loadCached(url).then(…)`, because the function is guaranteed to return a promise. That’s the purpose `Promise.resolve` in the line `(*)`: it makes sure the interface unified. We can always use `.then` after `loadCached`.

Promise.reject

The syntax:

```
let promise = Promise.reject(error);
```

Create a rejected promise with the `error`.

Same as:

```
let promise = new Promise((resolve, reject) => reject(error));
```

We cover it here for completeness, rarely used in real code.

Promise.all

The method to run many promises in parallel and wait till all of them are ready.

The syntax is:

```
let promise = Promise.all(iterable);
```

It takes an `iterable` object with promises, technically it can be any iterable, but usually it's an array, and returns a new promise. The new promise resolves with when all of them are settled and has an array of their results.

For instance, the `Promise.all` below settles after 3 seconds, and then its result is an array `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 3000)), // 1
  new Promise((resolve, reject) => setTimeout(() => resolve(2), 2000)), // 2
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 1000))  // 3
]).then(alert); // 1,2,3 when promises are ready: each promise contributes an array me
```

Please note that the relative order is the same. Even though the first promise takes the longest time to resolve, it is still first in the array of results.

A common trick is to map an array of job data into an array of promises, and then wrap that into `Promise.all`.

For instance, if we have an array of URLs, we can fetch them all like this:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// map every url to the promise fetch(github url)
let requests = urls.map(url => fetch(url));

// Promise.all waits until all jobs are resolved
Promise.all(requests)
  .then(responses => responses.forEach(
    response => alert(`${response.url}: ${response.status}`)
  ));
```

A more real-life example with fetching user information for an array of github users by their names (or we could fetch an array of goods by their ids, the logic is same):

```
let names = ['iliakan', 'remy', 'jeresig'];

let requests = names.map(name => fetch(`https://api.github.com/users/${name}`));

Promise.all(requests)
  .then(responses => {
    // all responses are ready, we can show HTTP status codes
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`); // shows 200 for every url
    }

    return responses;
  })
  // map array of responses into array of response.json() to read their content
  .then(responses => Promise.all(responses.map(r => r.json())))
  // all JSON answers are parsed: "users" is the array of them
  .then(users => users.forEach(user => alert(user.name)));
```

If any of the promises is rejected, `Promise.all` immediately rejects with that error.

For instance:

```
Promise.all([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).catch(alert); // Error: Whoops!
```

Here the second promise rejects in two seconds. That leads to immediate rejection of `Promise.all`, so `.catch` executes: the rejection error becomes the outcome of the whole `Promise.all`.

The important detail is that promises provide no way to “cancel” or “abort” their execution. So other promises continue to execute, and the eventually settle, but all their results are ignored.

There are ways to avoid this: we can either write additional code to `clearTimeout` (or otherwise cancel) the promises in case of an error, or we can make errors show up as members in the resulting array (see the task below this chapter about it).

`Promise.all(iterable)` allows non-promise items in `iterable`

Normally, `Promise.all(iterable)` accepts an iterable (in most cases an array) of promises. But if any of those objects is not a promise, it's wrapped in `Promise.resolve`.

For instance, here the results are `[1, 2, 3]`:

```
Promise.all([
  new Promise((resolve, reject) => {
    setTimeout(() => resolve(1), 1000)
  }),
  2, // treated as Promise.resolve(2)
  3  // treated as Promise.resolve(3)
]).then(alert); // 1, 2, 3
```

So we are able to pass non-promise values to `Promise.all` where convenient.

Promise.race

Similar to `Promise.all` takes an iterable of promises, but instead of waiting for all of them to finish – waits for the first result (or error), and goes on with it.

The syntax is:

```
let promise = Promise.race(iterable);
```

For instance, here the result will be `1`:

```
Promise.race([
  new Promise((resolve, reject) => setTimeout(() => resolve(1), 1000)),
  new Promise((resolve, reject) => setTimeout(() => reject(new Error("Whoops!")), 2000
  new Promise((resolve, reject) => setTimeout(() => resolve(3), 3000))
]).then(alert); // 1
```

So, the first result/error becomes the result of the whole `Promise.race`. After the first settled promise "wins the race", all further results/errors are ignored.

Summary

There are 4 static methods of `Promise` class:

1. `Promise.resolve(value)` – makes a resolved promise with the given value,

2. `Promise.reject(error)` – makes a rejected promise with the given error,
3. `Promise.all(promises)` – waits for all promises to resolve and returns an array of their results. If any of the given promises rejects, then it becomes the error of `Promise.all`, and all other results are ignored.
4. `Promise.race(promises)` – waits for the first promise to settle, and its result/error becomes the outcome.

Of these four, `Promise.all` is the most common in practice.

✓ Tasks

Fault-tolerant Promise.all

We'd like to fetch multiple URLs in parallel.

Here's the code to do that:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

Promise.all(urls.map(url => fetch(url)))
  // for each response show its status
  .then(responses => { // (*)
    for(let response of responses) {
      alert(`${response.url}: ${response.status}`);
    }
  });
```

The problem is that if any of requests fails, then `Promise.all` rejects with the error, and we loose results of all the other requests.

That's not good.

Modify the code so that the array `responses` in the line `(*)` would include the response objects for successful fetches and error objects for failed ones.

For instance, if one of URLs is bad, then it should be like:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'http://no-such-url'
];
```

```
Promise.all(...) // your code to fetch URLs...
  // ...and pass fetch errors as members of the resulting array...
  .then(responses => {
    // 3 urls => 3 array members
    alert(responses[0].status); // 200
    alert(responses[1].status); // 200
    alert(responses[2]); // TypeError: failed to fetch (text may vary)
  });
```

P.S. In this task you don't have to load the full response using `response.text()` or `response.json()`. Just handle fetch errors the right way.

Open a sandbox for the task.

To solution

Fault-tolerant fetch with JSON

Improve the solution of the previous task Fault-tolerant Promise.all. Now we need not just to call `fetch`, but to load the JSON objects from given URLs.

Here's the example code to do that:

```
let urls = [
  'https://api.github.com/users/iliakan',
  'https://api.github.com/users/remy',
  'https://api.github.com/users/jeresig'
];

// make fetch requests
Promise.all(urls.map(url => fetch(url)))
  // map each response to response.json()
  .then(responses => Promise.all(
    responses.map(r => r.json())
  ))
  // show name of each user
  .then(users => {  // (*)
    for(let user of users) {
      alert(user.name);
    }
  });
```

The problem is that if any of requests fails, then `Promise.all` rejects with the error, and we loose results of all the other requests. So the code above is not fault-tolerant, just like the one in the previous task.

Modify the code so that the array in the line `(*)` would include parsed JSON for successful requests and error for errored ones.

Please note that the error may occur both in `fetch` (if the network request fails) and in `response.json()` (if the response is invalid JSON). In both cases the error should become a member of the results object.

The sandbox has both of these cases.

Open a sandbox for the task.

To solution

Async/await

There’s a special syntax to work with promises in a more comfortable fashion, called “async/await”. It’s surprisingly easy to understand and use.

Async functions

Let’s start with the `async` keyword. It can be placed before a function, like this:

```
async function f() {
  return 1;
}
```

The word “async” before a function means one simple thing: a function always returns a promise. If the code has `return <non-promise>` in it, then JavaScript automatically wraps it into a resolved promise with that value.

For instance, the code above returns a resolved promise with the result of `1`, let’s test it:

```
async function f() {
  return 1;
}

f().then(alert); // 1
```

…We could explicitly return a promise, that would be the same:

```
async function f() {
  return Promise.resolve(1);
}

f().then(alert); // 1
```

So, `async` ensures that the function returns a promise, and wraps non-promises in it. Simple enough, right? But not only that. There's another keyword, `await`, that works only inside `async` functions, and it's pretty cool.

Await

The syntax:

```
// works only inside async functions
let value = await promise;
```

The keyword `await` makes JavaScript wait until that promise settles and returns its result.

Here's an example with a promise that resolves in 1 second:

```
async function f() {

  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000)
  });

  let result = await promise; // wait till the promise resolves (*)

  alert(result); // "done!"
}

f();
```

The function execution "pauses" at the line `(*)` and resumes when the promise settles, with `result` becoming its result. So the code above shows "done!" in one second.

Let's emphasize: `await` literally makes JavaScript wait until the promise settles, and then go on with the result. That doesn't cost any CPU resources, because the engine can do other jobs meanwhile: execute other scripts, handle events etc.

It's just a more elegant syntax of getting the promise result than `promise.then`, easier to read and write.

⚠ **Can't use `await` in regular functions**

If we try to use `await` in non-async function, that would be a syntax error:

```
function f() {
  let promise = Promise.resolve(1);
  let result = await promise; // Syntax error
}
```

We can get such error in case if we forget to put `async` before a function. As said, `await` only works inside `async function`.

Let's take `showAvatar()` example from the chapter Promises chaining and rewrite it using `async/await`:

1. We'll need to replace `.then` calls by `await`.
2. Also we should make the function `async` for them to work.

```
async function showAvatar() {

  // read our JSON
  let response = await fetch('/article/promise-chaining/user.json');
  let user = await response.json();

  // read github user
  let githubResponse = await fetch(`https://api.github.com/users/${user.name}`);
  let githubUser = await githubResponse.json();

  // show the avatar
  let img = document.createElement('img');
  img.src = githubUser.avatar_url;
  img.className = "promise-avatar-example";
  document.body.append(img);

  // wait 3 seconds
  await new Promise((resolve, reject) => setTimeout(resolve, 3000));

  img.remove();

  return githubUser;
}

showAvatar();
```

Pretty clean and easy to read, right? Much better than before.

`await` won't work in the top-level code

People who are just starting to use `await` tend to forget that, but we can't write `await` in the top-level code. That wouldn't work:

```
// syntax error in top-level code
let response = await fetch('/article/promise-chaining/user.json');
let user = await response.json();
```

So we need to have a wrapping async function for the code that awaits. Just as in the example above.

`await` accepts thenables

Like `promise.then`, `await` allows to use thenable objects (those with a callable `then` method). Again, the idea is that a 3rd-party object may not be a promise, but promise-compatible: if it supports `.then`, that's enough to use with `await`.

For instance, here `await` accepts `new Thenable(1)`:

```
class Thenable {
  constructor(num) {
    this.num = num;
  }
  then(resolve, reject) {
    alert(resolve); // function() { native code }
    // resolve with this.num*2 after 1000ms
    setTimeout(() => resolve(this.num * 2), 1000); // (*)
  }
};

async function f() {
  // waits for 1 second, then result becomes 2
  let result = await new Thenable(1);
  alert(result);
}

f();
```

If `await` gets a non-promise object with `.then`, it calls that method providing native functions `resolve`, `reject` as arguments. Then `await` waits until one of them is called (in the example above it happens in the line `(*)`) and then proceeds with the result.

Async methods

A class method can also be async, just put `async` before it.

Like here:

```
class Waiter {
  async wait() {
    return await Promise.resolve(1);
  }
}

new Waiter()
  .wait()
  .then(alert); // 1
```

The meaning is the same: it ensures that the returned value is a promise and enables `await`.

Error handling

If a promise resolves normally, then `await promise` returns the result. But in case of a rejection, it throws the error, just as if there were a `throw` statement at that line.

This code:

```
async function f() {
  await Promise.reject(new Error("Whoops!"));
}
```

…Is the same as this:

```
async function f() {
  throw new Error("Whoops!");
}
```

In real situations, the promise may take some time before it rejects. So `await` will wait, and then throw an error.

We can catch that error using `try..catch`, the same way as a regular `throw`:

```
async function f() {

  try {
    let response = await fetch('http://no-such-url');
  } catch(err) {
    alert(err); // TypeError: failed to fetch
  }
}

f();
```

In case of an error, the control jumps to the `catch` block. We can also wrap multiple lines:

```
async function f() {

  try {
    let response = await fetch('/no-user-here');
    let user = await response.json();
  } catch(err) {
    // catches errors both in fetch and response.json
    alert(err);
  }
}

f();
```

If we don't have `try..catch`, then the promise generated by the call of the async function `f()` becomes rejected. We can append `.catch` to handle it:

```
async function f() {
  let response = await fetch('http://no-such-url');
}

// f() becomes a rejected promise
f().catch(alert); // TypeError: failed to fetch // (*)
```

If we forget to add `.catch` there, then we get an unhandled promise error (and can see it in the console). We can catch such errors using a global event handler as described in the chapter Promises chaining.

`async/await` and `promise.then/catch`

When we use `async/await`, we rarely need `.then`, because `await` handles the waiting for us. And we can use a regular `try..catch` instead of `.catch`. That's usually (not always) more convenient.

But at the top level of the code, when we're outside of any `async` function, we're syntactically unable to use `await`, so it's a normal practice to add `.then/catch` to handle the final result or falling-through errors.

Like in the line `(*)` of the example above.

`async/await` works well with `Promise.all`

When we need to wait for multiple promises, we can wrap them in `Promise.all` and then `await`:

```
// wait for the array of results
let results = await Promise.all([
  fetch(url1),
  fetch(url2),
  ...
]);
```

In case of an error, it propagates as usual: from the failed promise to `Promise.all`, and then becomes an exception that we can catch using `try..catch` around the call.

Summary

The `async` keyword before a function has two effects:

1. Makes it always return a promise.
2. Allows to use `await` in it.

The `await` keyword before a promise makes JavaScript wait until that promise settles, and then:

1. If it's an error, the exception is generated, same as if `throw error` were called at that very place.
2. Otherwise, it returns the result, so we can assign it to a value.

Together they provide a great framework to write asynchronous code that is easy both to read and write.

With `async/await` we rarely need to write `promise.then/catch`, but we still shouldn't forget that they are based on promises, because sometimes (e.g. in the outermost scope) we have to use these methods. Also `Promise.all` is a nice thing to wait for many tasks simultaneously.

✓ Tasks

Rewrite "rethrow" async/await

Below you can find the "rethrow" example from the chapter Promises chaining. Rewrite it using `async/await` instead of `.then/catch`.

And get rid of the recursion in favour of a loop in `demoGithubUser`: with `async/await` that becomes easy to do.

```
class HttpError extends Error {
  constructor(response) {
    super(`${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new HttpError(response);
      }
    })
}

// Ask for a user name until github returns a valid user
function demoGithubUser() {
  let name = prompt("Enter a name?", "iliakan");

  return loadJson(`https://api.github.com/users/${name}`)
    .then(user => {
      alert(`Full name: ${user.name}.`);
      return user;
    })
    .catch(err => {
      if (err instanceof HttpError && err.response.status == 404) {
        alert("No such user, please reenter.");
        return demoGithubUser();
      } else {
        throw err;
      }
    });
}

demoGithubUser();
```

To solution

Rewrite using async/await

Rewrite the one of examples from the chapter Promises chaining using `async/await` instead of `.then/catch`:

```
function loadJson(url) {
  return fetch(url)
    .then(response => {
      if (response.status == 200) {
        return response.json();
      } else {
        throw new Error(response.status);
      }
    })
}

loadJson('no-such-user.json') // (3)
  .catch(alert); // Error: 404
```

To solution

Network requests: AJAX and COMET

XMLHttpRequest and AJAX

`XMLHttpRequest` is a built-in browser object that allows to make HTTP requests in JavaScript.

Despite of having the word "XML" in its name, it can operate on any data, not only in XML format.

Asynchronous XMLHttpRequest

XMLHttpRequest has two modes of operation: synchronous and asynchronous.

First let's see the asynchronous variant as it's used in the majority of cases.

The code below loads the URL at `/article/xmlhttprequest/hello.txt` from the server and shows its content on-screen:

```
// 1. Create a new XMLHttpRequest object
let xhr = new XMLHttpRequest();

// 2. Configure it: GET-request for the URL /article/.../hello.txt
xhr.open('GET', '/article/xmlhttprequest/hello.txt');

// 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
    // if it's not 200, consider it an error
    alert(xhr.status + ': ' + xhr.statusText); // e.g. 404: Not Found
  } else {
    // show the result
    alert(xhr.responseText); // responseText is the server response
  }
};
```

As we can see, there are several methods of `XMLHttpRequest` here. Let’s cover them.

Setup: “open”

The syntax:

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

This method is usually called first after `new XMLHttpRequest`. It specifies the main parameters of the request:

- `method` – HTTP-method. Usually `"GET"` or `"POST"`, but we can also use TRACE/DELETE/PUT and so on.
- `URL` – the URL to request. Can use any path and protocol, but there are cross-domain limitations called “Same Origin Policy”. We can make any requests to the same `protocol://domain:port` that the current page comes from, but other locations are “forbidden” by default (unless they implement special HTTP-headers, we’ll cover them in chapter [todo]).
- `async` – if the third parameter is explicitly set to `false`, then the request is synchronous, otherwise it’s asynchronous. We’ll talk more about that in this chapter soon.
- `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: “send”

The syntax:

```
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. We’ll see examples with a body in the next chapter.

Cancel: abort and timeout

If we changed our mind, we can terminate the request at any time. The call to `xhr.abort()` does that:

```
xhr.abort(); // terminate the request
```

We can also specify a timeout using the corresponding property:

```
xhr.timeout = 10000;
```

The timeout is expressed in ms. If the request does not succeed within the given time, it gets canceled automatically.

Events: onload, onerror etc

A request is asynchronous by default. In other words, the browser sends it out and allows other JavaScript code to execute.

After the request is sent, `xhr` starts to generate events. We can use `addEventListener` or `on<event>` properties to handle them, just like with DOM objects.

The modern specification lists following events:

- `loadstart` – the request has started.
- `progress` – the browser received a data packet (can happen multiple times).
- `abort` – the request was aborted by `xhr.abort()`.

- `error` – an network error has occurred, the request failed.
- `load` – the request is successful, no errors.
- `timeout` – the request was canceled due to timeout (if the timeout is set).
- `loadend` – the request is done (with an error or without it)
- `readystatechange` – the request state is changed (will cover later).

Using these events we can track successful loading (`onload`), errors (`onerror`) and the amount of the data loaded (`onprogress`).

Please note that errors here are “communication errors”. In other words, if the connection is lost or the remote server does not respond at all – then it’s the error in the terms of XMLHttpRequest. Bad HTTP status like 500 or 404 are not considered errors.

Here’s a more feature-full example, with errors and a timeout:

```
<script>
  function load(url) {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.timeout = 1000;
    xhr.send();

    xhr.onload = function() {
      alert(`Loaded: ${this.status} ${this.responseText}`);
    };

    xhr.onerror = () => alert('Error');

    xhr.ontimeout = () => alert('Timeout!');
  }
</script>

<button onclick="load('/article/xmlhttprequest/hello.txt')">Load</button>
<button onclick="load('/article/xmlhttprequest/hello.txt?speed=0')">Load with timeout<
<button onclick="load('no-such-page')">Load 404</button>
<button onclick="load('http://example.com')">Load another domain</button>
```

1. The first button triggers only `onload` as it loads the file `hello.txt` normally.
2. The second button loads a very slow URL, so it calls only `ontimeout` (because `xhr.timeout` is set).
3. The third button loads a non-existant URL, but it also calls `onload` (with “Loaded: 404”), because there’s no network error.
4. The last button tries to load a page from another domain. That’s prohibited unless the remote server explicitly agrees by sending certain headers (to be covered later), so we have `onerror` here. The `onerror` handler would also trigger in other cases if we start a request, and then sever the network connection of our device.

Response: status, responseText and others

Once the server has responded, we can receive the result in the following properties of the request object:

`status`

HTTP status code: `200`, `404`, `403` and so on. Also can be `0` if an error occurred.

`statusText`

HTTP status message: usually `OK` for `200`, `Not Found` for `404`, `Forbidden` for `403` and so on.

`responseText`

The text of the server response,

If the server returns XML with the correct header `Content-type: text/xml`, then there's also `responseXML` property with the parsed XML document. You can query it with `xhr.responseXml.querySelector("...")` and perform other XML-specific operations.

That's rarely used, because most of the time JSON is returned by the server. And then we can parse it using `JSON.parse(xhr.responseText)`.

Synchronous and asynchronous requests

If in the `open` method the third parameter `async` is set to `false`, the request is made synchronously.

In other words, Javascript execution pauses at that line and continues when the response is received. Somewhat like `alert` or `prompt` commands.

Synchronous calls are used rarely, because they block in-page Javascript till the loading is complete. In some browsers, a user is unable to scroll the page.

```
// Synchronous request
xhr.open('GET', 'phones.json', false);

// Send it
xhr.send();
// ...JavaScript "hangs" and waits till the request is complete
```

If a synchronous call takes too much time, the browser may suggest to close the "hanging" webpage.

Also, because of the blocking, it becomes impossible to send two requests simultaneously. And, looking a bit forward, let's note that some advanced capabilities of

`XMLHttpRequest`, like requesting from another domain or specifying a timeout, are unavailable for synchronous requests.

Because of all that, synchronous requests are used very sparingly, almost never.

By default, requests are asynchronous.

The same request made asynchronously:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', 'phones.json'); // the third parameter is true by default

xhr.send(); // (1)

xhr.onreadystatechange = function() { // (3)
  if (xhr.readyState != 4) return;

  button.innerHTML = 'Complete!';

  if (xhr.status != 200) {
    alert(xhr.status + ': ' + xhr.statusText);
  } else {
    alert(xhr.responseText);
  }

}

button.innerHTML = 'Loading...'; // (2)
button.disabled = true;
```

Now as there's no third argument in `open` (or if we explicitly set it to `true`), the request is asynchronous. In other words, after the call `xhr.send()` in the line `(1)`, Javascript does not “hang”, but continues to execute.

In our case, it means that `(2)` shows a “loading” message.

Then, after time, when the result is received, it comes in the event handler `(3)` that we'll cover a bit later.

Event “readystatechange”

The event `readystatechange` occurs multiple times during sending the request and receiving the response.

As the name suggests, there's a “ready state” of `XMLHttpRequest`. It is accessible as `xhr.readyState`.

In the example above we only used state `4` (request complete), but there are few more.

All states, as in the specification :

```
const unsigned short UNSENT = 0; // initial state
const unsigned short OPENED = 1; // open called
const unsigned short HEADERS_RECEIVED = 2; // response headers received
const unsigned short LOADING = 3; // response is loading (a data packed is received)
const unsigned short 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.

The example above demonstrates these states. The server answers the request `digits` by sending a string of `1000` digits once per second.

http://plnkr.co/edit/qM0vVFv4UpK0CEti1G5M?p=preview

Packets may break at any byte

One might think that `readyState=3` (the next data packet is received) allows us to get the current (not full yet) response body in `responseText`.

That’s true. But only partially.

Technically, we do not have control over breakpoints between network packets. Many languages use multi-byte encodings like UTF-8, where a character is represented by multiple bytes. Some characters use only 1 byte, some use 2 or more. And packets may split *in the middle of a character*.

E.g. the letter `ö` is encoded with two bytes. The first of them may be at the end of one packet, and the second one – at the beginning of the next packet.

So, during the `readyState`, at the end of `responseText` there will be a half-character byte. That may lead to problems. In some simple cases, when we use only latin characters and digits (they all are encoded with 1 byte), such thing can’t happen, but in other cases, that can become a source of bugs.

HTTP-headers

`XMLHttpRequest` allows both to send custom headers and read headers from the response.

There are 3 methods for HTTP-headers:

`setRequestHeader(name, value)`

Sets the request header with the given `name` and `value`.

For instance:

```
xhr.setRequestHeader('Content-Type', 'application/json');
```

⚠ Headers limitations

Several headers are managed exclusively by the browser, e.g. `Referer` and `Host`. The full list is in the specification.

XMLHttpRequest is not allowed to change them, for the sake of user safety and correctness of the request.

⚠ Can’t remove a header

Another peciliarity 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 acc;
  }, {});
```

Timeout

The maximum duration of an asynchronous request can be set using the `timeout` property:

```
xhr.timeout = 30000; // 30 seconds (in milliseconds)
```

If the request exceeds that time, it's aborted, and the `timeout` event is generated:

```
xhr.ontimeout = function() {
  alert( 'Sorry, the request took too long.' );
}
```

The full event list

The modern specification lists following events (in the lifecycle order):

- `loadstart` – the request has started.
- `progress` – a data packet of the response has arrived, the whole response body at the moment is in `responseText`.
- `abort` – the request was canceled by the call `xhr.abort()`.
- `error` – connection error has occured, 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` – the request has finished (succeffully or not).

The most used events are load completion (`onload`), load failure (`onerror`), and also `onprogress` to track the progress.

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.

Summary

Typical code of the GET-request with `XMLHttpRequest`:

```
let xhr = new XMLHttpRequest();

xhr.open('GET', '/my/url');

xhr.send();

xhr.onload = function() {
  // we can check
  // status, statusText - for response HTTP status
  // responseText, responseXML (when content-type: text/xml) - for the response

  if (this.status != 200) {
    // handle error
    alert( 'error: ' + this.status);
    return;
  }

  // get the response from this.responseText
};

xhr.onerror = function() {
  // handle error
};
```

XMLHttpRequest is widely used, but there's a more modern method named `fetch(url)` that returns a promise, thus working well with async/await. We'll cover it soon in the next sections.

Solutions

CSS-animations

Animate a plane (CSS)

CSS to animate both `width` and `height`:

```
/* original class */

#flyjet {
  transition: all 3s;
}

/* JS adds .growing */
#flyjet.growing {
  width: 400px;
  height: 240px;
}
```

Please note that `transitionend` triggers two times – once for every property. So if we don't perform an additional check then the message would show up 2 times.

Open the solution in a sandbox.

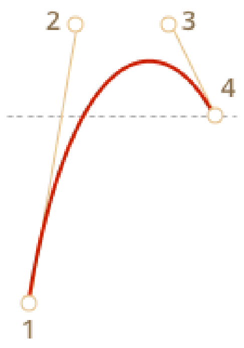
To formulation

Animate the flying plane (CSS)

We need to choose the right Bezier curve for that animation. It should have `y>1` somewhere for the plane to "jump out".

For instance, we can take both control points with `y>1`, like: `cubic-bezier(0.25, 1.5, 0.75, 1.5)`.

The graph:



Open the solution in a sandbox.

To formulation

Animated circle

Open the solution in a sandbox.

To formulation

JavaScript animations

Animate the bouncing ball

To bounce we can use CSS property `top` and `position:absolute` for the ball inside the field with `position:relative`.

The bottom coordinate of the field is `field.clientHeight`. But the `top` property gives coordinates for the top of the ball, the edge position is `field.clientHeight - ball.clientHeight`.

So we animate the `top` from `0` to `field.clientHeight - ball.clientHeight`.

Now to get the “bouncing” effect we can use the timing function `bounce` in `easeOut` mode.

Here’s the final code for the animation:

```
let to = field.clientHeight - ball.clientHeight;

animate({
  duration: 2000,
  timing: makeEaseOut(bounce),
  draw(progress) {
    ball.style.top = to * progress + 'px'
  }
});
```

Open the solution in a sandbox.

To formulation

Animate the ball bouncing to the right

In the task Animate the bouncing ball we had only one property to animate. Now we need one more: `elem.style.left`.

The horizontal coordinate changes by another law: it does not “bounce”, but gradually increases shifting the ball to the right.

We can write one more `animate` for it.

As the time function we could use `linear`, but something like `makeEaseOut(quad)` looks much better.

The code:

```
let height = field.clientHeight - ball.clientHeight;
let width = 100;

// animate top (bouncing)
animate({
  duration: 2000,
  timing: makeEaseOut(bounce),
  draw: function(progress) {
    ball.style.top = height * progress + 'px'
  }
});

// animate left (moving to the right)
animate({
  duration: 2000,
  timing: makeEaseOut(quad),
  draw: function(progress) {
    ball.style.left = width * progress + "px"
  }
});
```

Open the solution in a sandbox.

To formulation

Character classes

Find the time

The answer: `\b\d\d:\d\d\b`.

```
alert( "Breakfast at 09:00 in the room 123:456.".match( /\b\d\d:\d\d\b/ ) ); //
```

To formulation

Sets and ranges [...]

Java[^script]

Answers: **no, yes**.

- In the script `Java` it doesn't match anything, because `[^script]` means "any character except given ones". So the regexp looks for `"Java"` followed by one such symbol, but there's a string end, no symbols after it.

  ```
  alert( "Java".match(/Java[^script]/) ); // null
  ```

- Yes, because the regexp is case-insensitive, the `[^script]` part matches the character `"S"`.

  ```
  alert( "JavaScript".match(/Java[^script]/) ); // "JavaS"
  ```

To formulation

Find the time as hh:mm or hh-mm

Answer: `\d\d[-:]\d\d`.

```
let reg = /\d\d[-:]\d\d/g;
alert( "Breakfast at 09:00. Dinner at 21-30".match(reg) ); // 09:00, 21-30
```

Please note that the dash `'-'` has a special meaning in square brackets, but only between other characters, not when it's in the beginning or at the end, so we don't need to escape it.

To formulation

Quantifiers +, *, ? and {n}

How to find an ellipsis "..." ?

Solution:

```
let reg = /\.{3,}/g;
alert( "Hello!... How goes?.....".match(reg) ); // ..., .....
```

Please note that the dot is a special character, so we have to escape it and insert as `\.`.

To formulation

Regexp for HTML colors

We need to look for `#` followed by 6 hexadimal characters.

A hexadimal character can be described as `[0-9a-fA-F]`. Or if we use the `i` flag, then just `[0-9a-f]`.

Then we can look for 6 of them using the quantifier `{6}`.

As a result, we have the regexp: `/#[a-f0-9]{6}/gi`.

```
let reg = /#[a-f0-9]{6}/gi;

let str = "color:#121212; background-color:#AA00ef bad-colors:f#fddee #fd2"

alert( str.match(reg) );  // #121212,#AA00ef
```

The problem is that it finds the color in longer sequences:

```
alert( "#12345678".match( /#[a-f0-9]{6}/gi ) ) // #12345678
```

To fix that, we can add `\b` to the end:

```
// color
alert( "#123456".match( /#[a-f0-9]{6}\b/gi ) ); // #123456

// not a color
alert( "#12345678".match( /#[a-f0-9]{6}\b/gi ) ); // null
```

To formulation

Greedy and lazy quantifiers

A match for /d+? d+?/

The result is: `123 4`.

First the lazy `\d+?` tries to take as little digits as it can, but it has to reach the space, so it takes `123`.

Then the second `\d+?` takes only one digit, because that’s enough.

To formulation

Find HTML comments

We need to find the beginning of the comment `<!--`, then everything till the end of `-->`.

The first idea could be `<!--.*?-->` – the lazy quantifier makes the dot stop right before `-->`.

But a dot in Javascript means “any symbol except the newline”. So multiline comments won’t be found.

We can use `[\s\S]` instead of the dot to match “anything”:

```
let reg = /<!--[\s\S]*?-->/g;

let str = `... <!-- My -- comment
 test --> ..  <!----> ..
`;

alert( str.match(reg) ); // '<!-- My -- comment \n test -->', '<!---->'
```

To formulation

Find HTML tags

The solution is `<[^<>]+>`.

```
let reg = /<[^<>]+>/g;

let str = '<> <a href="/"> <input type="radio" checked> <b>';

alert( str.match(reg) ); // '<a href="/">', '<input type="radio" checked>', '<
```

To formulation

Capturing groups

Find color in the format #abc or #abcdef

A regexp to search 3-digit color `#abc`: `/#[a-f0-9]{3}/i`.

We can add exactly 3 more optional hex digits. We don't need more or less. Either we have them or we don't.

The simplest way to add them – is to append to the regexp: `/#[a-f0-9]{3}([a-f0-9]{3})?/i`

We can do it in a smarter way though: `/#([a-f0-9]{3}){1,2}/i`.

Here the regexp `[a-f0-9]{3}` is in parentheses to apply the quantifier `{1,2}` to it as a whole.

In action:

```
let reg = /#([a-f0-9]{3}){1,2}/gi;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(reg) ); // #3f3 #AA00ef #abc
```

There's a minor problem here: the pattern found `#abc` in `#abcd`. To prevent that we can add `\b` to the end:

```
let reg = /#([a-f0-9]{3}){1,2}\b/gi;

let str = "color: #3f3; background-color: #AA00ef; and: #abcd";

alert( str.match(reg) ); // #3f3 #AA00ef
```

To formulation

Find positive numbers

An integer number is `\d+`.

A decimal part is: `\.\d+`.

Because the decimal part is optional, let's put it in parentheses with the quantifier `'?'`.

Finally we have the regexp: `\d+(\.\d+)?`:

```
let reg = /\d+(\.\d+)?/g;

let str = "1.5 0 12. 123.4.";

alert( str.match(reg) );   // 1.5, 0, 12, 123.4
```

To formulation

Find all numbers

A positive number with an optional decimal part is (per previous task): `\d+(\.\d+)?`.

Let’s add an optional `-` in the beginning:

```
let reg = /-?\d+(\.\d+)?/g;

let str = "-1.5 0 2 -123.4.";

alert( str.match(reg) );   // -1.5, 0, 2, -123.4
```

To formulation

Parse an expression

A regexp for a number is: `-?\d+(\.\d+)?`. We created it in previous tasks.

An operator is `[-+*/]`. We put the dash `-` first, because in the middle it would mean a character range, we don’t need that.

Note that a slash should be escaped inside a JavaScript regexp `/.../`.

We need a number, an operator, and then another number. And optional spaces between them.

The full regular expression: `-?\d+(\.\d+)?\s*[-+*/]\s*-?\d+(\.\d+)?`.

To get a result as an array let’s put parentheses around the data that we need: numbers and the operator: `(-?\d+(\.\d+)?)\s*([-+*/])\s*(-?\d+(\.\d+)?)`.

In action:

```
let reg = /(-?\d+(\.\d+)?)\s*([-+*\/])\s*(-?\d+(\.\d+)?)/;

alert( "1.2 + 12".match(reg) );
```

The result includes:

- `result[0] == "1.2 + 12"` (full match)

- `result[1] == "1.2"` (first group `(-?\d+(\.\d+)?)` – the first number, including the decimal part)
- `result[2] == ".2"` (second group `(\.\d+)?` – the first decimal part)
- `result[3] == "+"` (third group `([-+*\/])` – the operator)
- `result[4] == "12"` (forth group `(-?\d+(\.\d+)?)` – the second number)
- `result[5] == undefined` (fifth group `(\.\d+)?` – the last decimal part is absent, so it's undefined)

We only want the numbers and the operator, without the full match or the decimal parts.

The full match (the arrays first item) can be removed by shifting the array `result.shift()`.

The decimal groups can be removed by making them into non-capturing groups, by adding `?:` to the beginning: `(?:\.\d+)?`.

The final solution:

```
function parse(expr) {
  let reg = /(-?\d+(?:\.\d+)?)\s*([-+*\/])\s*(-?\d+(?:\.\d+)?)/;

  let result = expr.match(reg);

  if (!result) return;
  result.shift();

  return result;
}

alert( parse("-1.23 * 3.45") );  // -1.23, *, 3.45
```

To formulation

Alternation (OR) |

Find programming languages

The first idea can be to list the languages with `|` in-between.

But that doesn't work right:

```
let reg = /Java|JavaScript|PHP|C|C\+\+/g;

let str = "Java, JavaScript, PHP, C, C++";

alert( str.match(reg) ); // Java,Java,PHP,C,C
```

The regular expression engine looks for alternations one-by-one. That is: first it checks if we have `Java`, otherwise – looks for `JavaScript` and so on.

As a result, `JavaScript` can never be found, just because `Java` is checked first.

The same with `C` and `C++`.

There are two solutions for that problem:

1. Change the order to check the longer match first: `JavaScript|Java|C\+\+|C|PHP`.
2. Merge variants with the same start: `Java(Script)?|C(\+\+)?|PHP`.

In action:

```
let reg = /Java(Script)?|C(\+\+)?|PHP/g;

let str = "Java, JavaScript, PHP, C, C++";

alert( str.match(reg) ); // Java,JavaScript,PHP,C,C++
```

To formulation

Find bbtag pairs

Opening tag is `\[(b|url|quote)\]`.

Then to find everything till the closing tag – let's the pattern `[\s\S]*?` to match any character including the newline and then a backreference to the closing tag.

The full pattern: `\[(b|url|quote)\][\s\S]*?\[/\1\]`.

In action:

```
let reg = /\[(b|url|quote)\][\s\S]*?\[\/\1\]/g;

let str = `
  [b]hello![/b]
  [quote]
    [url]http://google.com[/url]
  [/quote]
`;

alert( str.match(reg) ); // [b]hello![/b],[quote][url]http://google.com[/url][/
```

Please note that we had to escape a slash for the closing tag `[/\1]`, because normally the slash closes the pattern.

To formulation

Find quoted strings

The solution: `/"(\\.|[^"\\])*"/g`.

Step by step:

- First we look for an opening quote `"`
- Then if we have a backslash `\\` (we technically have to double it in the pattern, because it is a special character, so that's a single backslash in fact), then any character is fine after it (a dot).
- Otherwise we take any character except a quote (that would mean the end of the string) and a backslash (to prevent lonely backslashes, the backslash is only used with some other symbol after it): `[^"\\]`
- …And so on till the closing quote.

In action:

```
let reg = /"(\\.|[^"\\])*"/g;
let str = ' .. "test me" .. "Say \\"Hello\\"!" .. "\\\\ \\"" .. ';

alert( str.match(reg) ); // "test me","Say \"Hello\"!","\\ \""
```

To formulation

Find the full tag

The pattern start is obvious: `<style`.

…But then we can't simply write `<style.*?>`, because `<styler>` would match it.

We need either a space after `<style` and then optionally something else or the ending `>`.

In the regexp language: `<style(>|\s.*?>)`.

In action:

```
let reg = /<style(>|\s.*?>)/g;

alert( '<style> <styler> <style test="...">'.match(reg) ); // <style>, <style t
```

To formulation

String start ^ and finish $

Regexp ^$

The empty string is the only match: it starts and immediately finishes.

The task once again demonstrates that anchors are not characters, but tests.

The string is empty `""`. The engine first matches the `^` (input start), yes it's there, and then immediately the end `$`, it's here too. So there's a match.

To formulation

Check MAC-address

A two-digit hex number is `[0-9a-f]{2}` (assuming the `i` flag is enabled).

We need that number `NN`, and then `:NN` repeated 5 times (more numbers);

The regexp is: `[0-9a-f]{2}(:[0-9a-f]{2}){5}`

Now let's show that the match should capture all the text: start at the beginning and end at the end. That's done by wrapping the pattern in `^...$`.

Finally:

```
let reg = /^[0-9a-fA-F]{2}(:[0-9a-fA-F]{2}){5}$/i;

alert( reg.test('01:32:54:67:89:AB') ); // true

alert( reg.test('0132546789AB') ); // false (no colons)
alert( reg.test('01:32:54:67:89') ); // false (5 numbers, need 6)

alert( reg.test('01:32:54:67:89:ZZ') ) // false (ZZ in the end)
```

To formulation

Introduction: callbacks

Animated circle with callback

Open the solution in a sandbox.

To formulation

Promise

Re-resolve a promise?

The output is: `1`.

The second call to `resolve` is ignored, because only the first call of `reject/resolve` is taken into account. Further calls are ignored.

To formulation

Delay with a promise

```
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

delay(3000).then(() => alert('runs after 3 seconds'));
```

Please note that in this task `resolve` is called without arguments. We don't return any value from `delay`, just ensure the delay.

To formulation

Animated circle with promise

Open the solution in a sandbox.

To formulation

Promises chaining

Promise: then versus catch

The short answer is: **no, they are not the equal**:

The difference is that if an error happens in `f1`, then it is handled by `.catch` here:

```
promise
  .then(f1)
  .catch(f2);
```

…But not here:

```
promise
  .then(f1, f2);
```

That's because an error is passed down the chain, and in the second code piece there's no chain below `f1`.

In other words, `.then` passes results/errors to the next `.then/catch`. So in the first example, there's a `catch` below, and in the second one – there isn't, so the error is unhandled.

To formulation

Error in setTimeout

The answer is: **no, it won't**:

```
new Promise(function(resolve, reject) {
  setTimeout(() => {
    throw new Error("Whoops!");
  }, 1000);
}).catch(alert);
```

As said in the chapter, there's an "implicit `try..catch`" around the function code. So all synchronous errors are handled.

But here the error is generated not while the executor is running, but later. So the promise can't handle it.

To formulation

Promise API

Fault-tolerant Promise.all

The solution is actually pretty simple.

Take a look at this:

```
Promise.all(
  fetch('https://api.github.com/users/iliakan'),
  fetch('https://api.github.com/users/remy'),
  fetch('http://no-such-url')
)
```

Here we have an array of `fetch(...)` promises that goes to `Promise.all`.

We can't change the way `Promise.all` works: if it detects an error, then it rejects with it. So we need to prevent any error from occurring. Instead, if a `fetch` error happens, we need to treat it as a "normal" result.

Here's how:

```
Promise.all(
  fetch('https://api.github.com/users/iliakan').catch(err => err),
  fetch('https://api.github.com/users/remy').catch(err => err),
  fetch('http://no-such-url').catch(err => err)
)
```

In other words, the `.catch` takes an error for all of the promises and returns it normally. By the rules of how promises work, if a `.then/catch` handler returns a value (doesn't matter if it's an error object or something else), then the execution continues the "normal" flow.

So the `.catch` returns the error as a “normal” result into the outer `Promise.all`.

This code:

```
Promise.all(
  urls.map(url => fetch(url))
)
```

Can be rewritten as:

```
Promise.all(
  urls.map(url => fetch(url).catch(err => err))
)
```

Open the solution in a sandbox.

To formulation

Fault-tolerant fetch with JSON

Open the solution in a sandbox.

To formulation

Async/await

Rewrite "rethrow" async/await

There are no tricks here. Just replace `.catch` with `try...catch` inside `demoGithubUser` and add `async/await` where needed:

```
class HttpError extends Error {
  constructor(response) {
    super(`${response.status} for ${response.url}`);
    this.name = 'HttpError';
    this.response = response;
  }
}

async function loadJson(url) {
  let response = await fetch(url);
  if (response.status == 200) {
    return response.json();
  } else {
    throw new HttpError(response);
  }
}

// Ask for a user name until github returns a valid user
async function demoGithubUser() {

  let user;
  while(true) {
    let name = prompt("Enter a name?", "iliakan");

    try {
      user = await loadJson(`https://api.github.com/users/${name}`);
      break; // no error, exit loop
    } catch(err) {
      if (err instanceof HttpError && err.response.status == 404) {
        // loop continues after the alert
        alert("No such user, please reenter.");
      } else {
        // unknown error, rethrow
        throw err;
      }
    }
  }


  alert(`Full name: ${user.name}.`);
  return user;
}

demoGithubUser();
```

To formulation

Rewrite using async/await

The notes are below the code:

```
async function loadJson(url) { // (1)
  let response = await fetch(url); // (2)

  if (response.status == 200) {
    let json = await response.json(); // (3)
    return json;
  }

  throw new Error(response.status);
}

loadJson('no-such-user.json')
  .catch(alert); // Error: 404 (4)
```

Notes:

1. The function `loadUrl` becomes `async`.
2. All `.then` inside are replaced with `await`.
3. We can `return response.json()` instead of awaiting for it, like this:

   ```
   if (response.status == 200) {
     return response.json(); // (3)
   }
   ```

   Then the outer code would have to `await` for that promise to resolve. In our case it doesn't matter.
4. The error thrown from `loadJson` is handled by `.catch`. We can't use `await loadJson(…)` there, because we're not in an `async` function.

To formulation