link: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations/Using_CSS_animations

Using CSS animations

CSS animations make it possible to animate transitions from one CSS style configuration to another. Animations consist of two components, a style describing the CSS animation and a set of keyframes that indicate the start and end states of the animation's style, as well as possible intermediate waypoints.

There are three key advantages to CSS animations over traditional script-driven animation techniques:

1. They're easy to use for simple animations; you can create them without even having to know JavaScript.
2. The animations run well, even under moderate system load. Simple animations can often perform poorly in JavaScript. The rendering engine can use frame-skipping and other techniques to keep the performance as smooth as possible.
3. Letting the browser control the animation sequence lets the browser optimize performance and efficiency by, for example, reducing the update frequency of animations running in tabs that aren't currently visible.

Configuring an animation

To create a CSS animation sequence, you style the element you want to animate with the `animation` property or its sub-properties. This lets you configure the timing, duration, and other details of how the animation sequence should progress. This does **not** configure the actual appearance of the animation, which is done using the `@keyframes` at-rule as described in the Defining the animation sequence using keyframes section below.

The sub-properties of the `animation` property are:

`animation-delay`

Specifies the delay between an element loading and the start of an animation sequence and whether the animation should start immediately from its beginning or partway through the animation.

`animation-direction`

Specifies whether an animation's first iteration should be forward or backward and whether subsequent iterations should alternate direction on each run through the sequence or reset to the start point and repeat.

`animation-duration`

Specifies the length of time in which an animation completes one cycle.

`animation-fill-mode`

Specifies how an animation applies styles to its target before and after it runs.

`animation-iteration-count`

Specifies the number of times an animation should repeat.

`animation-name`

Specifies the name of the `@keyframes` at-rule describing an animation's keyframes.

`animation-play-state`

Specifies whether to pause or play an animation sequence.

`animation-timing-function`

Specifies how an animation transitions through keyframes by establishing acceleration curves.

Defining animation sequence using keyframes

After you've configured the animation's timing, you need to define the appearance of the animation. This is done by establishing one or more keyframes using the `@keyframes` at-rule. Each keyframe describes how the animated element should render at a given time during the animation sequence.

Since the timing of the animation is defined in the CSS style that configures the animation, keyframes use a `<percentage>` to indicate the time during the animation sequence at which they take place. 0% indicates the first moment of the animation sequence, while 100% indicates the final state of the animation. Because these two times are so important, they have special aliases: `from` and `to`. Both are optional. If `from` / `0%` or `to` / `100%` is not specified, the browser starts or finishes the animation using the computed values of all attributes.

You can optionally include additional keyframes that describe intermediate steps between the start and end of the animation.

Using the animation shorthand

The `animation` shorthand is useful for saving space. As an example, some of the rules we've been using through this article:

```
p {
  animation-duration: 3s;
  animation-name: slidein;
  animation-iteration-count: infinite;
  animation-direction: alternate;
}
```

...could be replaced by using the `animation` shorthand.

```
p {
  animation: 3s infinite alternate slidein;
}
```

To learn more about the sequence in which different animation property values can be specified using the `animation` shorthand, see the `animation` reference page.

Setting multiple animation property values

The CSS animation longhand properties can accept multiple values, separated by commas. This feature can be used when you want to apply multiple animations in a single rule and set different durations, iteration counts, etc., for each of the animations. Let's look at some quick examples to explain the different permutations.

In this first example, there are three duration and three iteration count values. So each animation is assigned a value of duration and iteration count with the same position as the animation name. The `fadeInOut` animation is assigned a duration of `2.5s` and an iteration count of `2`, and the `bounce` animation is assigned a duration of `1s` and an iteration count of `5`.

```
animation-name: fadeInOut, moveLeft300px, bounce;
animation-duration: 2.5s, 5s, 1s;
animation-iteration-count: 2, 1, 5;
```

In this second example, three animation names are set, but there's only one duration and iteration count. In this case, all three animations are given the same duration and iteration count.

```
animation-name: fadeInOut, moveLeft300px, bounce;
animation-duration: 3s;
animation-iteration-count: 1;
```

In this third example, three animations are specified, but only two durations and iteration counts. In such cases where there are not enough values in the list to assign a separate one to each animation, the value assignment cycles from the first to the last item in the available list and then cycles back to the first item. So, `fadeInOut` gets a duration of `2.5s`, and `moveLeft300px` gets a duration of `5s`, which is the last value in the list of duration values. The duration value assignment now resets to the first value; `bounce`, therefore, gets a duration of

`2.5s`. The iteration count values (and any other property values you specify) will be assigned in the same way.

```
animation-name: fadeInOut, moveLeft300px, bounce;
animation-duration: 2.5s, 5s;
animation-iteration-count: 2, 1;
```

If the mismatch in the number of animations and animation property values is inverted, say there are five `animation-duration` values for three `animation-name` values, then the extra or unused animation property values, in this case, two `animation-duration` values, don't apply to any animation and are ignored.

Examples

> **Note:** Some older browsers (pre-2017) may need prefixes; the live examples you can click to see in your browser include the `-webkit` prefixed syntax.

Making text slide across the browser window

This simple example styles the `<p>` element so that the text slides in from off the right edge of the browser window.

Note that animations like this can cause the page to become wider than the browser window. To avoid this problem put the element to be animated in a container, and set `overflow :hidden` on the container.

```
p {
  animation-duration: 3s;
  animation-name: slidein;
}

@keyframes slidein {
  from {
    margin-left: 100%;
```

```
    width: 300%;
  }

  to {
    margin-left: 0%;
    width: 100%;
  }
}
```

In this example the style for the `<p>` element specifies that the animation should take 3 seconds to execute from start to finish, using the `animation-duration` property, and that the name of the `@keyframes` at-rule defining the keyframes for the animation sequence is named "slidein".

If we wanted any custom styling on the `<p>` element to appear in browsers that don't support CSS animations, we would include it here as well; however, in this case we don't want any custom styling other than the animation effect.

The keyframes are defined using the `@keyframes` at-rule. In this case, we have just two keyframes. The first occurs at 0% (using the alias `from`). Here, we configure the left margin of the element to be at 100% (that is, at the far right edge of the containing element), and the width of the element to be 300% (or three times the width of the containing element). This causes the first frame of the animation to have the header drawn off the right edge of the browser window.

The second (and final) keyframe occurs at 100% (using the alias `to`). The left margin is set to 0% and the width of the element is set to 100%. This causes the header to finish its animation flush against the left edge of the content area.

```
<p>
  The Caterpillar and Alice looked at each other for some time in silence: at
  last the Caterpillar took the hookah out of its mouth, and addressed her in a
  languid, sleepy voice.
</p>
```

> **Note:** Reload page to see the animation.

The Caterpillar and Alice looked at each other for some time in silence: at last the Caterpillar took the hookah out of its mouth, and addressed her in a languid, sleepy voice.

Adding another keyframe

Let's add another keyframe to the previous example's animation. Let's say we want the header's font size to increase as it moves from right to left for a while, then to decrease back to its original size. That's as simple as adding this keyframe:

```
75% {
  font-size: 300%;
  margin-left: 25%;
  width: 150%;
}
```

The full code now looks like this:

```
p {
  animation-duration: 3s;
  animation-name: slidein;
}

@keyframes slidein {
  from {
```

```
    margin-left: 100%;
    width: 300%;
  }

  75% {
    font-size: 300%;
    margin-left: 25%;
    width: 150%;
  }

  to {
    margin-left: 0%;
    width: 100%;
  }
}

<p>
  The Caterpillar and Alice looked at each other for some time in silence: at
  last the Caterpillar took the hookah out of its mouth, and addressed her in a
  languid, sleepy voice.
</p>
```

This tells the browser that 75% of the way through the animation sequence, the header should have its left margin at 25% and the width should be 150%.

> **Note:** Reload page to see the animation.

The Caterpillar and Alice looked at each other for some time in silence: at last the Caterpillar took the hookah out of its mouth, and addressed her in a languid, sleepy voice.

Repeating the animation

To make the animation repeat itself, use the `animation-iteration-count` property to indicate how many times to repeat the animation. In this case, let's use `infinite` to have the animation repeat indefinitely:

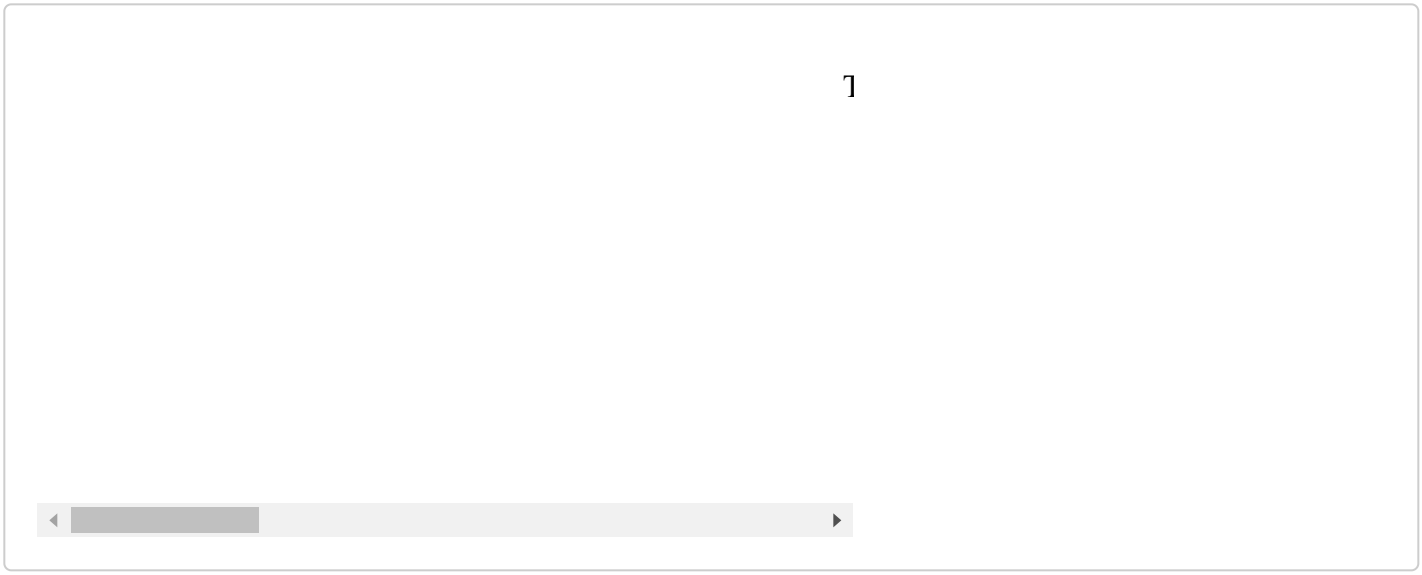
```
p {
  animation-duration: 3s;
  animation-name: slidein;
  animation-iteration-count: infinite;
}
```

Adding it to the existing code:

```
@keyframes slidein {
  from {
    margin-left: 100%;
    width: 300%;
  }

  to {
    margin-left: 0%;
    width: 100%;
  }
}
```

```
<p>
  The Caterpillar and Alice looked at each other for some time in silence: at
  last the Caterpillar took the hookah out of its mouth, and addressed her in a
  languid, sleepy voice.
</p>
```



Making the animation move back and forth

That made it repeat, but it's very odd having it jump back to the start each time it begins animating. What we really want is for it to move back and forth across the screen. That's easily accomplished by setting `animation-direction` to `alternate`:

```
p {
  animation-duration: 3s;
  animation-name: slidein;
  animation-iteration-count: infinite;
  animation-direction: alternate;
}
```

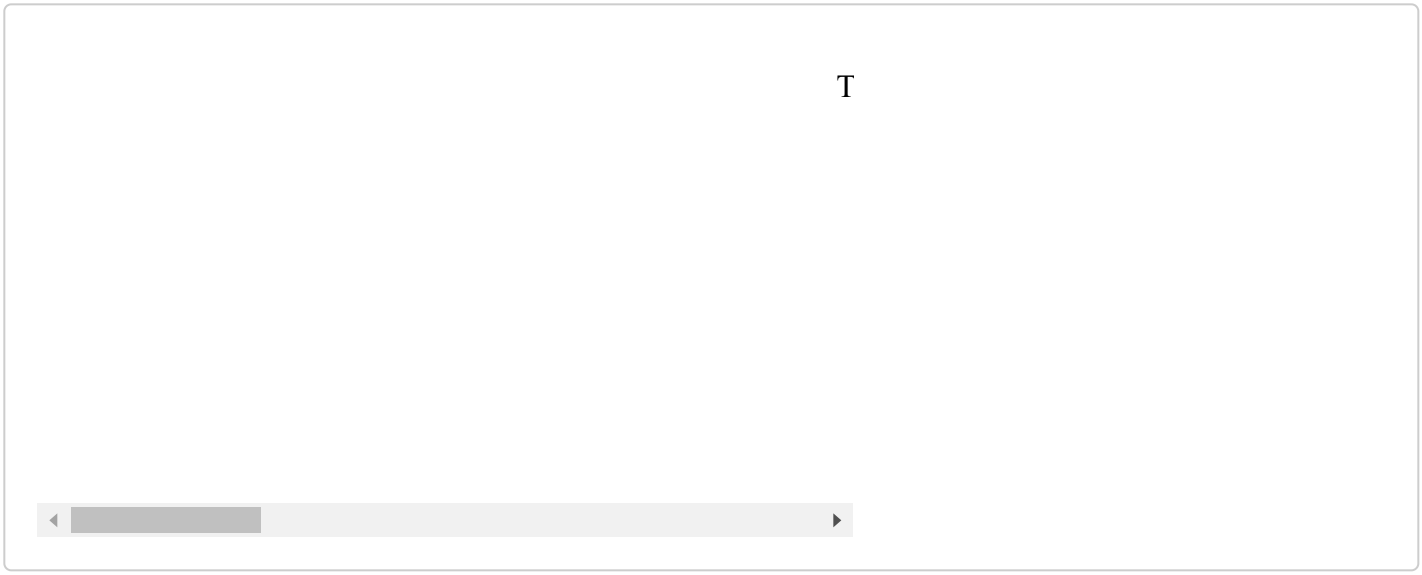
And the rest of the code:

```
@keyframes slidein {
  from {
    margin-left: 100%;
```

```
    width: 300%;
  }

  to {
    margin-left: 0%;
    width: 100%;
  }
}

<p>
  The Caterpillar and Alice looked at each other for some time in silence: at
  last the Caterpillar took the hookah out of its mouth, and addressed her in a
  languid, sleepy voice.
</p>
```



Using animation events

You can get additional control over animations — as well as useful information about them — by making use of animation events. These events, represented by the `AnimationEvent` object, can be used to detect when animations start, finish, and begin a new iteration. Each event includes the time at which it occurred as well as the name of the animation that triggered the event.

We'll modify the sliding text example to output some information about each animation event when it occurs, so we can get a look at how they work.

Adding the CSS

We start with creating the CSS for the animation. This animation will last for 3 seconds, be called "slidein", repeat 3 times, and alternate direction each time. In the `@keyframes`, the width and margin-left are manipulated to make the element slide across the screen.

```
.slidein {
  animation-duration: 3s;
  animation-name: slidein;
  animation-iteration-count: 3;
  animation-direction: alternate;
}

@keyframes slidein {
  from {
    margin-left: 100%;
    width: 300%;
  }

  to {
    margin-left: 0%;
    width: 100%;
  }
}
```

Adding the animation event listeners

We'll use JavaScript code to listen for all three possible animation events. This code configures our event listeners; we call it when the document is first loaded in order to set things up.

```
const element = document.getElementById("watchme");
element.addEventListener("animationstart", listener, false);
element.addEventListener("animationend", listener, false);
element.addEventListener("animationiteration", listener, false);

element.className = "slidein";
```

This is pretty standard code; you can get details on how it works in the documentation for `eventTarget.addEventListener()`. The last thing this code does is set the `class` on the element we'll be animating to "slidein"; we do this to start the animation.

Why? Because the `animationstart` event fires as soon as the animation starts, and in our case, that happens before our code runs. So we'll start the animation ourselves by setting the class of the element to the style that gets animated after the fact.

Receiving the events

The events get delivered to the `listener()` function, which is shown below.

```
function listener(event) {
  const l = document.createElement("li");
  switch(event.type) {
    case "animationstart":
      l.textContent = `Started: elapsed time is ${event.elapsedTime}`;
      break;
    case "animationend":
      l.textContent = `Ended: elapsed time is ${event.elapsedTime}`;
      break;
    case "animationiteration":
      l.textContent = `New loop started at time ${event.elapsedTime}`;
      break;
  }
  document.getElementById("output").appendChild(l);
}
```

This code, too, is very simple. It looks at the `event.type` to determine which kind of animation event occurred, then adds an appropriate note to the `<ul>` (unordered list) we're using to log these events.

The output, when all is said and done, looks something like this:

- Started: elapsed time is 0
- New loop started at time 3.01200008392334

- New loop started at time 6.00600004196167
- Ended: elapsed time is 9.234000205993652

Note that the times are very close to, but not exactly, those expected given the timing established when the animation was configured. Note also that after the final iteration of the animation, the `animationiteration` event isn't sent; instead, the `animationend` event is sent.

Just for the sake of completeness, here's the HTML that displays the page content, including the list into which the script inserts information about the received events:

```
<h1 id="watchme">Watch me move</h1>
<p>
  This example shows how to use CSS animations to make <code>H1</code>
  elements move across the page.
</p>
<p>
  In addition, we output some text each time an animation event fires, so you
  can see them in action.
</p>
<ul id="output"></ul>
```

And here's the live output.

> **Note:** Reload page to see the animation.

Watch me move

This example shows how to use CSS animations to make `H1` elements move across the page.

In addition, we output some text each time an animation event fires, so you can see them in action.

- Started: elapsed time is 0
- Ended: elapsed time is 9

See also

mdn web docs_

Last modified: Sep 28, 2022, by MDN contributors