X

Animation

CSS Transitions enable simple animations, but the start and end states of the animation are controlled by existing property values and provide little to no control over how the animation progresses.

Animations are similar to transitions in that values of CSS properties change over time. But unlike transitions, where we can only use timing functions to control how the animation goes from the original value to the final value, CSS keyframe animations enable us to granularly control what happens throughout the animation. With transitions, an element's properties change from the value set in one style block to the values set in a different style block as the element changes state over time instead of instantly had no transition been set. CSS animations are different.With animations, you can change property values that are not part of a pre or post state of the animated element. The property values set on the animated element don't necessarily have to be part of the animation progression. With transitions, going from black to white will only display varying shades of grey. With animation, that same element doesn't have to be black or white or inbetween during the animation. While you can stick with only grays, you could change the colors and animate from yellow to orange, or you could animate thru various colors, starting with black, ending with white, but progressing thru the entire rainbow if you so choos. With animations you can use as many keyframes as you want to create the desired effect.

While transitions trigger implicit property values changes, animations are explicitly executed when the animation keyframe properties are applied.

CSS animation enables us to animate the values of CSS properties over time, using keyframes. Similar to transitions, animation provides us with greater control over the duration, number of repeats, the repeating behavior, and what happens before the first animation commences and after the last animation iteration concludes. CSS animation properties allow us to control the timing, and even pause an animation mid-stream.

The first step in creating an animation is creating a keyframe animation. The keyframe animation is a @keyframes at-rule that defines which properties will be animated and how. The second step is to apply the keyframe animation you created to one or more elements in your document or application, using various animation properties to define how that animation will progress the keyframes.

Keyframes

To animate, within a selector's CSS block we provide, at minimum, the name and duration of a keyframe animation. We first define this CSS keyframe animation separately using the `@keyframes` at-rule.

To define our CSS animation, we declare its keyframes using the `@keyframes` rule. We give our animation a name. The name we create will then be used within a CSS selector's code block to attach this particular animation to the elements and pseudo elements defined by the selector.

The `@keyframes` at-rule includes the animation identifier and one or more keyframe blocks. Each keyframe block includes one or more keyframe selectors and a declaration block of zero or more property / value pairs. The entire at-rule specifies the behavior of one full iteration of the animation. The animation may iterate one or more times, or even less than one time.

```
@keyframes animation_identifier {
  keyframe_selectorA {
    property1: value1a;
    property2: value2b;
  }
  keyframe_selectorB {
    property1: value1b;
    property2: value2b;
  }
}
```

The animation_identifier is the name you give your animation for future reference. The specific rules for choosing a valid name are described below. The animation is enclosed in curly braces, with a series of keyframe blocks.

Each keyframe block includes one or more keyframe selectors, which are the percentage in time along the duration of the animation the block's property values should be in effect. The keyframe selectors are declared either as percentages or the keyterms `from` or `to`. The keyframe selectors are time positions along the duration of the animation. Each keyframe block, which includes zero or more property value pairs, is enclosed in curly braces.

Setting up your keyframe animation

We create a `@keyframes` at-rule, with an animation name, and a series of keyframe selectors with blocks of CSS in which you declare the properties we want to animate at that percentage of the way thru the animation. The keyframes we declare don't in themselves animate anything. Rather, we must attach the keyframe animations we created via the `animation-name` property, whose value is the name we provided within our at-rule. We discuss the `animation-name` property below.

Start with the at-rule declaration, followed by a name you create, and brackets:

```
@keyframes the_name_you want {
...
}
```

The name, which you create, is an identifier, not a string. Identifiers have specific rules. First, they can't be quoted. You can use any characters [a-zA-Z0-9], the hyphen (-), underscore (*), and any ISO 10646 character U+00A0 and higher. ISO 10646 is the universal character set. This means you can use any character in the unicode standard that matches the regular expression [*-a-zA-Z0-9\u00A0-\u10FFFF].

There are some limitations on the name. As mentioned above, do not quote the animation identifier (or animation name). The name can't start with a digit [0-9], two hyphens, though one hyphen is fine as long as it is not followed by a digit, unless you escape the digit or dash with a back slash. Make sure to escape them any escape characters. For example, Q&A must be written as Q&A!. âœŽ can be left as âœŽ, but if you are going to use any keyboard characters that aren't letters or digits, like !, @, #, $, %, ^, &, *, (,), +, =, ~, `, ,, ., ', ", ;, :, [,], {, }, I, \ and /, escape it with a back slash.

Also, don't use any of the keyterms covered in this chapter as the name for your animation. For example, possible values for the various animation properties we'll be covering later in the chapter include `none`, `paused`, `running`, `infinite`, `backwards`, and `forwards`, among others. Using an animation property keyterm, while not prohibited by the spec, will likely break your animation when using the `animation` shorthand property discussed below, or even the `animation-name` property in the case you called your animation `none`.

After declaring the name of our @keyframe animation, we encompass all the rules of our at-rule in curly braces. This is where we will put all our keyframes.

Keyframe Selectors

Keyframes selectors provide points during our animation where we set the values of the properties we want to animate.

The keyframe selectors consist of a comma-separated list of one or more percentage values or the keywords `from` or `to`. The keyword `from` is equal to `0%`. The keyword `to` equals `100%`. The selectors are used to specify the percentage along the duration of the animation the keyframe represents. The keyframe itself is specified by the block of property values declared on the selector. The `%` unit must be used on percentage values: in other words, `0` is invalid as a keyframe selector.

```
@keyframes W {
    from {
      left: 0;
      top: 0;
    }
    25%, 75% {
      top: 100%;
    }
    50% {
      top: 50%;
    }
    to {
      left: 100%;
      top: 0;
    }
}
```

In the above example, which would move a relatively or absolutely positioned element along a W shaped path if applied, has 5 keyframes at the 0%, 25%, 50%, 75% and 100% mark. The `from` is the 0% mark, the `to` is the 100% mark, and, as the property values we set for both the 25% and 75% mark is the same, we put two together as a comma-separated list of keyframe selectors.

Note selectors do not need to be listed in order. In the above example, we've placed the 25% and 75% on the same line, with the 50% mark coming after that declaration. For ease of legibility, it is highly encouraged to progress from the 0% to the 100% mark, but, as demonstrated by the "out of order" 75% keyframe above, it is not required.

If a `0%` or `from` keyframe is not specified, then the user agent constructs a `0%` keyframe using the original values of the properties being animated: as if the 0% keyframe were declared with the property values when no animation was applied. Similarly, if the `100%` or `to` keyframe is not defined, the browser creates a faux `100%` keyframe using the value the element would have had had no animation been set on it.

Assuming we have a background color change animation:

```
@keyframes change_bgcolor {
45% { background-color: green; }
55% { background-color: blue; }
}
```

And the element originally had it's `background-color` set to red, it would be as if the animation were written as:

```
@keyframes change_bgcolor {
    _0%   { background-color: red;}_
    45%  { background-color: green; }
    55%  { background-color: blue; }
    _100% { background-color: red;}_
}
```

or, remembering that we can including multiple, identical keyframes as a comma separated list remembering the background-color: red declarations are not actually part of the keyframe animation. If the background color were set to yellow in the element's default state, the 0% and 100% marks would display a yellow background, animating into green then blue, then back to yellow as the animation progressed:

```
@keyframes change_bgcolor {
    _0%, 100% { background-color: red;}_
    45% { background-color: green; }
    55% { background-color: blue; }
}
```

Negative percentages, values greater than `100%` and values that aren't otherwise percentages or the keyterms `to` or `from` are not valid and will be ignored.

In the original -webkit- implementation of animation, each keyframe could only be declared once: if declared more than once, only the last declaration would be applied, and the previous keyframe selector block was ignored. This has been updated. Now, similar to the rest of CSS, the values in the keyframe declaration blocks with identical keyframe values cascade. In the standard (non prefixed) syntax, the `W` animation above can be written with the `to`, or `100%`, declared twice, overriding the value of the left property:

```
@keyframes W {
  from, to {
    top: 0;
    left: 0;
  }
  25%, 75% {
    top: 100%;
  }
  50% {
    top: 50%;
  }
  to {
    left: 100%;
  }
}
```

Note in the above code block, the `to` is declared along with `from` as keyframe selectors for the first code block. The `left` value is overwritten for the `to` in the last keyframe block.

Only animate animatable properties. Like the rest of CSS, properties and values in a keyframe declaration block that are not understood, are ignored. Properties that are not animatable are also ignored (with the exception of `animation-timing-function`). There is a fairly exhaustive list of animatable properties maintained by the community on the Mozilla Developer Network, including which property values are animatable.

> Note: The `animation-timing-function`, described in greater detail below, while not an animatable property, is not ignored. If you include the `animation-timing-function` as a keyframe style rule within a keyframe selector block, the `animation-timing-function` will change to the declared timing function when the animation moves to the next keyframe.

Other than a few exceptions, do not try to animate between non-numeric values. You can animate between values that are written in a non-numeric way, as long as they can be extrapolated into a numeric value, like named colors which are extrapolated to hexadecimal color values.

If the animation is set between two property values that don't have a mid-point the results may not be what you expect: the property will not animated correctly or at all. For example, you can't animate between `height: auto;` and `height: 300px;`. There is no mid-point between `auto` and `300px`. The element may still animate, but different browsers handle this differently: Firefox does not animate the element. Safari may animate as if `auto` is equal to `0`, and both Opera and Chrome currently jump from the pre animated state to the post animated state half way through the animation, which may or may not be at the 50% keyframe selector, depending on the value of the `animation-timing-function`.

Different browsers behave in differently for different properties when there is no midpoint. The behavior of your animation will be most predictable if you declare both a 0% and a 100% for every property you animate. For example, if you declare `border-radius: 50%;` in your animation, you may want to declare `border-radius: 0;` as-well, because there is no mid-point between `none` and anything: the default value of `border-radius` is `none`, not `0`.

```
@keyframes round {
    100% {
        border-radius: 50%;
    }
}
```

The above will animate an element using the original value of the `border-radius` of that element to a border-radius of 50% over the duration of the animation cycle. If no `border-radius` value was set on the element upon which the animation got applied, the default value of `border-radius: none` will be used, which may lead to an effect you did not expect.

```
@keyframes square_to_round {
    0% {
        border-radius: 0%;
    }
    100% {
        border-radius: 50%;
    }
}
```

While including a 0% keyframe will ensure that your animation runs smoothly, the element may have had rounded corners to begin with. By adding `border-radius: 0%;` in the `from` keyframe, if the element was originally rounded, it will jump to rectangular corners before it starts animating. This might not be what you want. The best way to resolve this issue is to use the *round* animation instead of *square_to_round*, but make sure any element that gets animated with the *round* keyframe animation has its `border-radius` explicitly set.

Exceptions to the midpoint rule include `visibility` and `animation-timing-function`. `Visibility` is an animatable property. When you animate from `hidden` to `visible` the effect is that it jumps from one value to the next at the keyframe upon which it is declared. While the `animation-timing-function` is not in fact an animatable property, when included in a keyframe block, it will jump to the new value when the keyframe that declares the new value is reached. This is discussed below.

That being said, not all the properties need to be included in each keyframe block. As long as an animatable property is included in at-least one block with a value that is different then the non-animated attribute value, and there is a possible midpoint between those two values, that property will animate.

Animated Elements

Once you have created a keyframe animation, you need to apply that animation to an element or pseudo element for anything to actually animate. CSS Animation provides us with numerous animation properties to attach a keyframe animation to an element and control its progression. At minimum, we need to include the name of the animation for element to animate, and a duration if we want the animation to actually be visible.

There are three animation events -- the `animationstart`, `animationend` and `animationiteration` DOM events -- that occur at the start and end of an animation, and between the end of an iteration and the start of the next iteration. Any animation for which a valid keyframe rule is defined will generate the start and end events, even animations with empty keyframe rules. The animationiteration event only occurs when an animation has more than one iteration, as the `animationiteration` even does not fire if the `animationend` event is firing at the same time.

The `animation-name` property

The `animation-name` property takes as it's value the name or comma-separated names of the keyframe animation(s) you want to apply to that element. The names are the unquoted identifiers you created in your @keyframes rule.

animation-name

Values:

```
<@keyframes_identifier> | none | inherit | initial
```

Initial value:

```
none
```

Applies to:

```
all elements, ::before and ::after pseudo-elements
```

Inherited:

```
No
```

The default value is `none`, which means there is no animation. The `none` value can be used to override any animation applied elsewhere in the CSS cascade. To apply an animation, include the @keyframe identifier.

```
    div {
        animation-name: change_bgcolor;
    }
```

To apply more than one animation, include more than one comma-separated @keyframe identifiers.

```
    div {
        animation-name: change_bgcolor, round, W;
    }
```

If one of the included keyframe identifiers does not exist, the series of animation will not fail: rather, the failed animation will be ignored, the valid animations will be applied. While ignored initially, the failed animation will be applied if and when that identifier does come into existence as a valid animation.

To include more than one animation you must include each @keyframe animation identifier as a list of comma separated values on the `animation-name` property. If more than one animation is applied to an element that have repeated properties, the latter animations override the property values in the preceding animations. See Animation, Specificity and Precedence Order.

If you include three animation names, all the following properties such as `animation-duration` and `animation-iteration-count` should have three values as well, so that there are corresponding values for each attached animation. If there are too many values, the extra values are ignored. If there are too few comma separated values, the provided values will be repeated.

> If an included keyframe identifiers doesn't exist, the animation doesn't fail. Any other animations attached via the `animation-name` property will proceed normally. If that non-existant animation comes into existence, the animation will be attached to that element when the identifier becomes valid and will start iterating immediately or after the expiration of any `animation-delay`.

Simply applying an animation to an element is not enough for the element to visibly animate. For an element to visibly animate, the animation must last at least some amount of time. For that we have the `animation-duration` property.

Simply applying an animation, however, will get the `animationstart` and `animationend` events to occur. A single `animationstart` event occurs when the an animation starts and a single `animationend` event occurs when the animation ends, even if there are no perceptible animation occurs

The `animation-duration` property`

The `animation-duration` property defines how long a single animation iteration should take in seconds or milliseconds.

animation-duration

Values:

```
<time>
```

Initial value:

```
0s
```

Applies to:

```
all elements, ::before and ::after pseudo-elements
```

Inherited:

```
No
```

The `animation-duration` property takes as it's value the length of time, in seconds (s) or milliseconds (ms), it should take to complete one cycle thru all the keyframes. If omitted, the animation will still be applied with a duration of 0s, with `animationstart`, `animationend` and `animationinteration` still being fired even though the animation, taking `0s`, is imperceptible. Negative values are invalid, and will behave as if the default of `0s` were applied.

Generally, you will want to include an `animation-duration` value for each `animation-name` provide. If you have only one duration, all the animations will last the same amount of time. Having fewer `animation-duration` values than `animation-name` values in your comma-separated property value list will not fail: rather, the values that are included will be repeated. If you have a greater number of `animation-duration` values than `animation-name` values, the extra values will be ignored.

The `animation-iteration-count` property

Include the `animation-iteration-count` property if you want to iterate through the animation more than the default one time. By default, simply including the required `animation-name`, the animation will play once.

animation-iteration-count

Values:

```
&lt;number> | infinite
```

Initial value:

```
1
```

Applies to:

```
all elements, ::before and ::after pseudo-elements
```

Inherited:

```
No
```

By default, the animation will occur once. If included, the animation will repeat the number of times specified by the value if the `animation-iteration-count` property, which can be any number or the keyterm `infinite`. If the number is not an integer, the animation will end partway through its last cycle. The animation will still run, but will cut off mid iteration on the final iteration. For example, `animation-iteration-count: 1.25` will iterate thru the animation 1.25 times, cutting off 25% thru the second iteration. If the value is 0.25 on an 8 second animation, the animation will play 25% of the way thru, ending after 2 seconds. Negative numbers are not valid, leading to a default single iteration.

Interestingly, 0 is a valid value for the `animation-iteration-count` property When set to 0, the animation still occurs over 0s, similar to setting `animation-duration: 0s;`, throwing both an `animationstart` and an `animationend` event. If the `animation-iteration-count` is omitted, or set to a value of 1 or less, the `animationiteration` event will not occur.

```
.flag {
    animation-name: red, white, blue;
    animation-duration: 2s, 4s, 6s;
    animation-iteration-count: 3, 5;
}
```

If you are attaching more than one animation to an element or pseudo-element, include a comma separated list of values for `animation-name`, `animation-duration` and `animation-iteration-count`. The iteration count values, and all other animation property values, will be assigned to the animation in the order of the comma-separated `animation-name` property value. Extra values will be ignored. Missing values will cause the existing values to be repeated. In the above example, red and blue will go thru 3 cyles, and white will iterate 5 times.

If we wanted all three animations to end at the same time, even though their durations differ, we can control that with `animation-iteration-count`.

```
.flag {
    animation-name: red, white, blue;
    animation-duration: 2s, 4s, 6s;
    animation-iteration-count: 6, 3, 2;
}
```

In the above example, the red, white and blue animations will last for a total of 12 seconds each: red lasts 2s, iterating 6 times, for a total of 12 seconds, white lasts 4s, iterating 3 times, for a total of 12 seconds, and blue lasts 6s, iterating 2 times, for a total of 12 seconds. With simple arithmetic you can figure out how many iterations you need to make one effect last as long as another, remembering that the `animation-iteration-count` value doesn't need to be an integer.

Note: The `animationiteration` event only fires *between* iterations, so no `animationiteration` event will be fired unless an animation cycle ends and another begins. A missing `animation-iteration-count` property, or any value of one or less, means no `animationiteration` event will be fired. As long as an iteration finishes, and the next iteration starts, even if neither is a full iteration, then the `animationiteration` event will occur, but not otherwise.

The `animation-direction` property

With the `animation-direction` property you can control whether the animation progresses from the 0% keyframe to the 100% keyframe, or from the 100% keyframe to the 0% keyframe. You can control whether all the iterations progress in the same direction or set every other animation cycle to progress in the reverse direction.

animation-iteration-direction

Values:

```
normal | reverse | alternate | alternate-reverse
```

Initial value:

```
normal
```

Applies to:

```
all elements, ::before and ::after pseudo-elements
```

Inherited:

```
No
```

The `animation-direction` property defines the direction of the progression of the animation thru the keyframes. There are four possible values:

`animation-iteration-direction: normal`

When set to `normal`, each iteration of the animation progresses from the 0% keyframe to the 100% keyframe.

`animation-iteration-direction: reverse`

The `reverse` value sets each iteration to play in reverse keyframe order, always progressing from the 100% keyframe to the 0% keyframe.

`animation-iteration-direction: alternate`

The `alternate` value means that the first iteration (and each subsequent odd numbered iteration) should proceed from 0% to 100%, the second iteration (and each subsequent even numbered cycle) should reverse direction, proceeding from 100% to 0%.

`animation-iteration-direction: alternate-reverse`

The `alternate-reverse` value is similar to the `alternate` value, except the odd numbered iterations are in the reverse direction, and the even numbered animation iterations are in the normal, or 0% to 100%; direction.

```
.ball {
    animation-name: bouncing;
    animation-duration: 400ms;
    animation-iteration-count: infinite;
    animation-direction: alternate-reverse;
    animation-timing-function: ease-out;
}
@keyframes bouncing {
    from {
        transforms: translateY(500px);
    }
    to {
        transforms: translateY(0);
    }
}
```

When an animation is played in reverse the timing function is reversed. In the above example, we are bouncing our ball, but we want to start it before it drops, and we want it to alternate between going up and going down, so `animation-direction: alternate-reverse;` is the most appropriate value for our needs. An important thing to note here is that when the ball is dropping, it gets faster. As it goes up, it gets slower: the ball is slowest at the apex and fastest at the nadir. When played in reverse, our `ease-out` animation appears to be `ease-in` animation, progressing from the 100% keyframe to the 0% keyframe. The `animation-timing-function`, along with the `ease-in` and `ease-out` values are described below.

The `animation-delay` property

The `animation-delay` property defines how long the browser waits after the animation is attached to the element before beginning the first animation iteration. The default value is 0s, meaning the animation will commence immediately when it is applied. A positive value will delay the start of the animation until the prescribed time listed as the value of the `animation-delay` property has elapsed. A negative value will cause the animation to begin immediately part way thru the animation.

animation-delay

Values:

```
\<time>
```

Initial value:

```
0s
```

Applies to:

```
all elements, ::before and ::after pseudo-elements
```

Inherited:

```
No
```

The time, defined in seconds (s) or milliseconds (ms) defines the delay between when the animation is attached to the element and when the animation begins executing. By default, the animation begins iterating as soon as it is applied to the element, with a 0s delay.

Including a negative value for the `animation-delay` property is valid, and can create interesting effects. A negative delay will execute the animation immediately, but will begin animating the element part way thru the attached animation. For example, if `animation-delay: -4s;` and `animation-duration: 10s;` are set on an element, the animation will begin immediately, but will start 40% of the way thru the first animation, which is not necessarily the 40% keyframe block: this depends on the value of the `animation-timing-function`.

```
div {
  animation-name: move;
  animation-duration: 10s;
  animation-delay: -4s;
  animation-timing-function: linear;
}
@keyframes move {
  from { transform: translateX(0); }
  to { transform: translateX(1000px); }
}
```

In the above example, for a `linear` animation, the animation would start with the `div` translated 400px to the right of its original position. The `animation-timing-function` is described below.

If an animation is set to occur 10 times, with a delay of -4s, with an animation iteration lasting 1s, the element will start animating right away, at the beginning of the fifth iteration.

```
.ball {
  animation-name: bounce;
  animation-duration: 1s;
  animation-delay: -4s;
  animation-iteration-count: 10;
  animation-timing-function: ease-in;
  animation-direction: alternate;
}
@keyframes bounce {
  from { transform: translateY(0); }
  to { transform: translateY(500px); }
}
```

Instead of animating for 10 seconds, the ball will animate for 6 seconds, with the animation starting immediately, but at the start of the 5th iteration, so in the reverse direction as the `animation-direction` is set to alternate, meaning every odd iteration iterates in the reverse direction from 100% keyframe to the 0% keyframe. It will throw the `animationstart` immediately, with have 5 `animationiteration` events. The `animationend` will occur at the 6 second mark.

The `animation-timing-function` property

Similar to the `transition-timing-function` property, the `animation-timing-function` property describes how the animation will progress over one cycle of its duration.

animation-timing-function

Values:

```
ease | linear | ease-in | ease-out | |ease-in-out | step-start |
 step-end | steps(&lt;integer>, start) | steps(&lt;integer>, end
) | cubic-bezier(&lt;number>, &lt;number>, &lt;number>, &lt;numb
er>)
```

Initial value:

```
ease
```

Applies to:

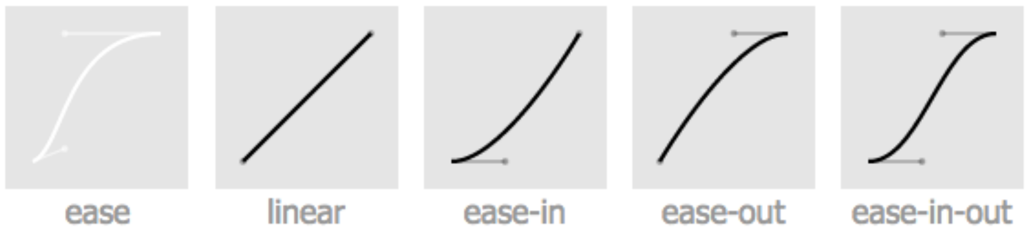
```
all elements, ::before and ::after pseudo-elements
```

Inherited:

```
No
```

Other than the step timing functions, described below, the timing functions are keyterms associated with predefined Bézier curves or you can define your own Bézier curve. Bézier curves are mathematically defined parametric curves used in two-dimensional graphic applications. The curve is defined by four points: the initial position and the terminating position (which are called "anchors") and two separate middle points (which are called "handles"). In CSS, the anchors are at 0, 0 and 1, 1.

Yes, you can define your own Bézier curve, and there are 5 pre-defined Bézier curves: `ease`, `linear`, `ease-in`, `ease-out` and `ease-in-out`.

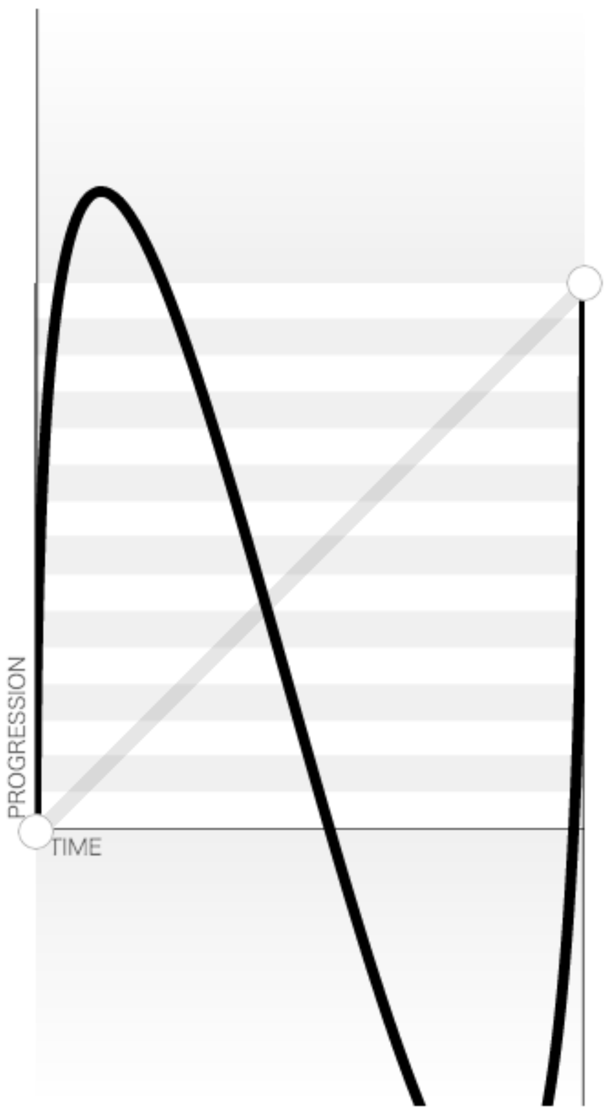


The default `ease` is equal to `cubic-bezier(0.25, 0.1, 0.25, 1)`, which has a slow start, then it speeds up, and ends slowly. This function is similar to `ease-in-out` at `cubic-bezier(0.42, 0, 0.58, 1)`, though it accelerates more sharply at the beginning. `linear` is equal to `cubic-bezier(0, 0, 1, 1)`, and, as the name describes, creates an animation that animates at a constant speed.. `ease-in` is equal to `cubic-bezier(0.42, 0, 1, 1)`, which creates an animation that is slow to start, but gains speed, then stops abruptly. The opposite `ease-out` timing function is equal to `cubic-bezier(0, 0, 0.58, 1)`, starting and full speed, then slowing progressively as it reaches the conclusion of the animation iteration. If none of these work for you, you can create your own bezier curve timing function by passing 4 values:

```
animation-timing-function: cubic-bezier(0.2, 0.4, 0.6, 0.8);
```

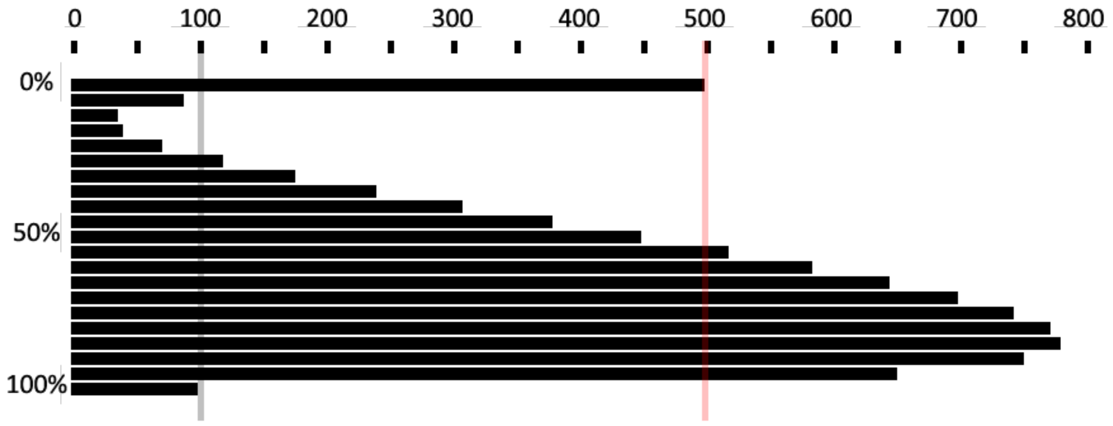
A Bézier curve takes four values. The first two at the x and y of the first point on the curve, and the last two are the x and y of the second point on the curve. The x values must between 0 and 1, or the Bézier curve is invalid. When creating your own Bézier curve, remember the steeper the curve the faster the motion. The flatter the curve, the slower the motion.

By using values for Y that are greater than 1 or less than 0, you can create a bouncing effect, making the animation bounced up and down between values, rather than going consistently in a single direction.



```
.snake {
  animation: shrink 10s cubic-bezier(0, 4, 1, -4) 2s both;
}
@-webkit-keyframes shrink {
  0% {width: 500px;}
  100% {width: 100px;}
}
```

The above `animation-timing-function` value makes the property values go outside the boundaries of the 0% and 100% keyframes. In this example we are shrinking an element from 500px to 100px. However, because of the `cubic-bezier` values, the element we're shrinking will actually grow to be wider than the 500px defined in the 0% keyframe, and smaller than 100px defined in the 100% keyframe.



In this scenario, the snake sits the width at 500px, the 0% keyframe, thru the expiration of the animation delay. It then quicky shrinks down to about 25px wide, which is smaller than the 100px declared in the 100% keyframe, before slowly expanding to about 750px wide, which is larger than the original 500px width. It then quickly shrinks back down to 100px, which is the value defined in the 100% keyframe, staying there because the `animation-fill-mode` value is set to `both`.

You may note the curve created by our animation is the same curve as our Bézier curve. Just like our s-curve goes below and above our bounding box, the width of our animation goes narrower than the 100px minimum width set and wider than the 500px maximum width we set up.

The Bézier curve has the appearance of a snake, going up and down and up again because one Y coordinate is positive and the other negative. If both are positive values greater than 1 or both negative less than -1, the Bézier curve is arced shaped, going above or below one value, but not bouncing like the s-curve above.

The timing declared for the `animation-timing-function` is the timing for the normal animation direction, when the animation is progressing from the 0% to 100% mark. When the animation is running in the reverse direction, from the 100% to the 0% mark, the animation timing function is inverted.

```
.ball {
      animation-name: bounce;
      animation-duration: 1s;
      animation-iteration-count: infinite;
      animation-timing-function: ease-in;
      animation-direction: alternate;
    }
    @keyframes bounce {
      0% { transform: translateY(0); }
      100% { transform: translateY(500px); }
    }
```

If we remember our bouncing ball example from above, when the ball is dropping it gets faster as it nears its nadir at the 100% keyframe, with the animation-timing-function set to `ease-in`. When it is bouncing up, it is animating in the reverse direction, from 100% to 0%, so the `animating-timing-function` is reversed as well, to `ease-out`, slowing down as it reaches its apex.

The step timing functions, `step-start`, `step-end` and `steps()`, are the animation timing functions that are not cubic bezier curves.

The `steps()` timing function divides the animation into a series of equal length steps. `steps()` takes two parameters: the number of steps, and the direction.

The number of steps is the first parameter. The value must be a positive integer. The animation will be divided equally into the number of steps provided. For example, if the animation duration is 1 second, and the number of steps is 5, the animation will be divided into five 200ms steps, with the element being redrawn to the page 5 times, at 200ms intervals, moving 20% thru the animation at each interval.

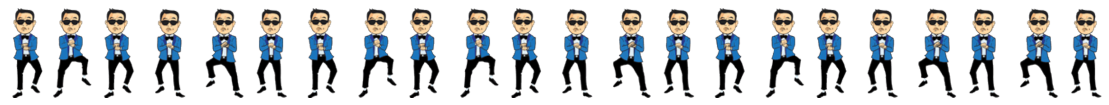
If the animation does that, it either draws the animation at the 0%, 20%, 40%, 60%, and 80% keyframes or at the 20%, 40%, 60%, 80% and 100% keyframes. It will either skip drawing the 100% or the 0% keyframe. That is where the direction parameter comes in.

The *direction* parameter takes one of two values: either `start` or `end`. The direction determines if the function is left- or right-continuous: basically, if the 0% or the 100% keyframe is going to be skipped. Including `start` as the second parameter will create a left-continuous function, meaning the first step happens when the animation begins, skipping the 0%, but including the 100%. Including `end`, or omitting a second parameter (`end` is the default direction) will create a right-continuous function. This mean the first step will be at the 0% mark, and the last step will be before the 100% mark. With `end`, the 100% keyframe will not be seen unless `animation-timing-function` of either `forwards` or `both` is set.

The `step-start` value is equal to `steps(1, start)`, with only a single step being the 100% keyframe. The `step-end` value is equal to `steps(1, end)`, which displays only the 0% keyframe.

The `steps()` function is most useful when it comes to character animation: for a simple game of pong, the cubic-bezier timing functions suffice. But, if you want to animate complex shapes that subtley change, like the drawings or pictures in a flip book, the `steps()` timing function is the solution.

A flip book is a book with a series of pictures, each containing a single drawing or picture, that vary gradually from one page to the next: like one frame from a movie or cartoon stamped onto each page. When the pages of a flip book are flipped thru rapidly (hence the name), the pictures appear as an animated motion. You can create similar animations with CSS using an image sprite, the `background-position` property and the `steps()` timing function.



In our image sprite we have several images that change just slightly: like the drawings on the individual pages of our flip book.

We put all of our slightly differing images into a single image: this image is called a 'sprite'. Each image in our sprite is a frame in the single animated image we're creating.

We create a container element that is the size of a single image of our sprite. We attach the sprite as the background image of our container element. We then animate the `background-position`, using the `steps()` timing function so we only see a single instance of the changing image of our sprite at a time. The number of steps in our `steps()` timing function is one more than the number of occurrences of the image in our sprite. The number of steps defines how many stops our background image makes to complete a single animation.

Our sprite has 22 images, for a total size of 100px by 1232px. That means each of our images within the sprite is 100px tall by 56px wide. We set our container to that size: 56px X 100px. We set our sprite as our background image: by default the image will appear at 0 0, which is a good default for older browsers that don't support CSS animation.

```
.dancer {
    height: 100px;
    width: 56px;
    background-image: url(../images/dancer.png);
    ....
}
```

The trick is to use `steps()` to change the `background-position` value so each frame is a view of a separate image within the sprite. Instead of sliding in the background image from the left, the `steps()` timing function will pop in the background image in the number of steps we declared.

We declare our animation to simply be a change in the background position. The image is 1230px: so we move the background image from the top left (0px from the top and 0px from the left), to negative 1230px. While -1230px will move the image completely out to the left, no longer showing as a background image in our 100px by 56px div at the 100% mark unless background-repeat repeats it, we remember that with the `steps(n, end)` syntax, the 100% keyframe never gets hit as the animation runs. Had we used `start`, the 0% keyframe wouldn't show. As we used `end`, the 100% keyframe doesn't show. This is what we want.

```
@keyframes danceinplace {
    from {
        background-position: 0 0;
    }
    to {
        background-position: -1230px
    }
}

.dancer {
    ....
    background-image: url(../images/dancer.png);
    animation: danceinplace 1s steps(22, end) infinite;
}
```

We used `steps(22, end)`. We use the `end` direction to show the 0% keyframe but not the 100% keyframe. What may have seemed like a complex animation is very simple: just like a flip book we see one frame of the sprite at a time. Our keyframe animation simply moves the background.

Animating the `animation-timing-function`

The `animation-timing-function` is not an animatable property: it won't slowly change from one value to another if included within keyframe selector blocks. However, unlike other animation properties, it does have an effect when specified on individual keyframes.

```
@keyframes bouncing {
     40%, 70%, 90%, 100%{
        bottom: 0;
        animation-timing-function: ease-out;
    }
    0%, 55%, 80%, 95% {
        animation-timing-function: ease-in;
    }
    0% {
        bottom: 200px;
        left: 0;
    }
    55% {
        bottom: 50px;
    }
    80% {
        bottom: 25px;
    }
    95% {
        bottom: 10px;
    }
    100% {
        left: 110px;
    }
}
```

Bouncing balls accelerate as they fall, and decelerate as they go upwards.

Under `animation-direction`, and then again above, we learned `animation-timing-functions` invert when the animation is proceeding in the reverse direction. However, the bouncing ball we created bounced forever. In reality, the ball will lose momentum when gravitational potential energy is converted to kinetic energy: the ball will eventually stop bouncing. That animation, therefore, was really not realistic.

We need to create an animation that loses height after a few bounces, and eventually stops. Our more realistic animation will have a single iteration, with the granular control provided via the keyframe blocks. Instead of relying on the animation-direction to change our timing function, we need to ensure that with each bounce the ball loses momentum within our keyframe blocks. We still want it to speed up with gravity, and slow down as it reaches it's apex. Because we will have only a single iteration, we have to control the timing within our keyframes. This is doable.

In our bouncing keyframe animation, `animation-timing-function` changes from `ease-in` to `ease-out` when the ball bounces, and from `ease-out` to `ease-in` at the apexes. The `animation-timing-function` property isn't animated in the sense of changing from one value to another over time. Rather, it changes from one value to the next when the it reaches a keyframe selector declaring a change to that value.

Specifying the `animation-timing-function` within the `to` or `100%` keyframe will have not effect on the animation.

The `animation-play-state` property

The `animation-play-state` property defines whether the animation is running or paused.

animation-play-state

Values:

```
running | paused
```

Initial value:

```
running
```

Applies to:

```
all elements, ::before and ::after pseudo-elements
```

Inherited:

```
No
```

When set to the default `running`, the animation proceeds as normal. If set to `paused`, the animation will be paused. When paused, the animation is still applied to the element, paused at the progress it had made before being paused. When unpaused (set back to running), it restarts from where it left off, as if the "clock" that controls the animation had stopped and started again.

If the property is set to `animation-play-state: paused` during the delay phase of the animation, the delay clock is also paused and resumes expiring as soon as `animation-play-state` is set back to running.

The `animation-fill-mode` property

The `animation-fill-mode` property defines what values are applied by the animation before and after the animation iterations are executed, enabling us to define if an element's property values are applied by the animation outside of the animation executing, before the `animationstart` event and / or after the `animationend` events are fired.

By default, an animation will not affect property values of the element on which it is attached until after the animation delay has expired up until the last iteration has completed. We can control this with the `animation-fill-mode` property. The `animation-fill-mode` property enables us to apply the property values of the first keyframe to an element as soon as the animation is applied to that element thru the animation delay. It also enables us to maintain the property values of the last keyframe after the last animation cycle is complete.

animation-fill-mode

Values:

```
none | forwards | backwards | both
```

Initial value:

```
none
```

Applies to:

```
all elements, ::before and ::after pseudo-elements
```

Inherited:

```
No
```

The default value is `none`, which means the animation has no effect when it is not executing: the animation's 0% keyframe block property values are not applied to the animated element until the `animation-delay` has expired, until the `animationstart` event is fired.

When the value is set to `backwards`, the property values from the 0% keyframe will be applied to the element as soon as the animation is applied to the element. The 0% keyframe property values are applied immediately (or 100% keyframe in the case of `reversed` or `reversed-alternate` value for the `animation-direction` property), without waiting for the `animation-delay` time to expire, before the `animationstart` event fires.

The value of `forwards` means when the animation is done executing -- has concluded the last part of the last iteration as defined by the `animation-iteration-count` value - it continues to apply the values of the properties at the values they were at when the `animationend` event occured. This is either the 100% keyframe, or, if the last iteration was in the reverse direction, the 0% keyframe. If the `animation-iteration-count` was not an integer, this will be somewhere in between the 0% and 100% keyframes.

The value of `both` applies both the `backwards` effect of applying the property values when

If the `animation-iteration-count` is a float value, and not an integer, the last iteration will not end on the 0% or 100% keyframe: the animation will end its execution partway through an animation cycle. If the If the `animation-fill-mode` was set `forwards` or `both` the element will maintain the property values it had when the `animationend` event occured.

For example, if we take the following code:

```
@keyframes moveMe {
  0% {
    transform: translatex(0);
  }
  100% {
    transform: translatex(1000px);
  }
}
.moved {
  animation: moveMe 10s linear 0.6 forwards;
}
```

The animation will only go thru 0.6 iterations. Being a linear, 10 second animation, it will stop at the 60% mark 6 seconds into the animation, when the element is translated 600px to the right. With `animation-fill-mode` set to `forwards` or `both`, the animation will stop animating when it is translated 600px to the right, holding the moved element 600px to the right of it's original position, keeping it translated indefinitely, or until the animation is detached from the element.

In Safari 8 and earlier, the element will jump to the values set in the 100% keyframe. In this case, it will jump from being translated by 400px to be 1000px to the right of where it normally would have been, and stay there indefinitely or until the animation is detached from the moved element. In Safari, it doesn't matter whether the last iteration was `normal` or `reverse`, or whether the animation ended 25% or 75% thru an animation cycle, `animation-fill-mode: forward;` causes the animation to jump to the 100% frame, and stay there, no matter where the partial animation concluded. This follows an older version of the specification, but we expect it will be updated to match the updated specification and all other evergreen browsers.

The `backwards` value controls what happens to the element after the animation was attached to the element, and up until the animation delay expires and the animation starts executing. Before the animation starts executing (during the period specified by animation-delay), the animation applies the values that it will start the first iteration with. The values specified in the animation's 0% keyframe are applied if the animation-direction is normal or alternate immediately when the animation is attached. If the animation-direction is reverse or alternate-reverse, the property values of the 100% keyframe are used.

If the 0% or 100% keyframes are not explicitly defined, the browser uses the implied values for those keyframes.

The value of `both` simply means that both `forwards` and `backwards` fill will be applied. As soon as the animation is attached to an element, that element will assume the properties provided in the 0% keyframe (or 100% keyframe in `animation-direction` is set to `reverse` or `alternate-reverse`). When the last iteration concludes, it will be as if the `animation-fill-mode` were set to `forwards`: if it was a full iteration in the normal direction, the property values of the 100% keyframe will be applied. If the last cycle was in the reverse direction, the property values of the 0% keyframe will be applied. If it wasn't a full iteration, the values that were present when the `animationend` event occurred will stay in effect.

If the `animation-duration` is set to 0s, no matter the `animation-iteration-count`, even if the count is `infinite`, if `backward` or `both` is set, the animation will stay on the 0% keyframe (or 100% keyframe if `animation-direction` is set to `reverse` or `reverse-alternate`) until the animation delay has expired, and will immediately jump to the 100% keyframe (or the 0% keyframe if `animation-direction` is set to `reverse` or `reverse-alternate`), staying on that keyframe in perpetuity, or until the animation is removed from the element or generated content. In this case, the `animationstart` and `animationend` events will occur in succession at the expiration of the delay, and, even if the iteration count is 100, there will be no `animationiteration` event.

The `animation` shorthand property

The `animation` shorthand property enables us to use one line instead of eight to define all the animation properties on an element. The `animation` property values are space separated, with the animation shorthand being a comma-separated list of space separated animation properties.

animation

Values:

```
none | <series of individual animation properties>

<animation-duration> || <animation-timing-function> || <animatio
n-delay> || <animation-iteration-count> || <animation-direction>
 || <animation-fill-mode> || <animation-play-state> || <animatio
n-name>
```

Initial value:

```
0s ease 0s 1 normal none running none
```

Applies to:

```
all elements, ::before and ::after pseudo-elements
```

Inherited:

```
No
```

The animation shorthand takes as it value all the other animation properties above, including `animation-duration`, `animation-timing-function`, `animation-delay`, `animation-iteration-count`, `animation-direction`, `animation-fill-mode`, `animation-play-state`, and `animation-name`.

```
#animated {
    animation: 200ms ease-in 50ms forwards slidedown;
}
```

is the equivalent of:

```
#animated {
    animation-name: slidedown;
    animation-duration: 200ms;
    animation-timing-function: ease-in;
    animation-delay: 50ms;
    animation-iteration-count: 1;
    animation-fill-mode: forwards;
    animation-direction: normal;
    animation-play-state: running;
}
```

or

```
#animated {
    animation: 200ms ease-in 50ms 1 normal running forwards slid
edown;
}
```

We didn't have to declare all of the values in the animation shorthand: any values that aren't declared are set to the default values. The above line is long, and in this case, 3 of the properties are set to default, so are not necessary.

It's important to remember that if you don't declare all 8 values, the ones you don't declare will get the default value for that property. The default values are:

```
animation-name: none;
animation-duration: 0s;
animation-timing-function: ease;
animation-delay: 0;
animation-iteration-count: 1;
animation-fill-mode: none;
animation-direction: normal;
animation-play-state: running;
```

The order of the shorthand is partially important. For example, there are two time properties: the first is always interpreted as the duration, the second, if present, is interpreted as the delay.

The placement of the `animation-name` can also be important. If you use an animation property value as your animation identifier, which you shouldn't, the `animation-name` should be placed as the *last* property value in the `animation` shorthand. The first occurrence of a keyword that is a valid value for any property other than `animation-name`, such as `ease`, and `running`, will be assumed to be part of the shorthand of the animation property they're normally associated with rather than the `animation-name`. If you observe the initial value at the beginning of this section you'll note `none` appears twice: the first is therefore the value for `animation-fill-mode`, the second is the value of the `animation-name`.

```
#failedAnimation {
    animation: paused 2s;
}
```

The above is the equivalent to

```
 #failedAnimation {
    animation-name: none;
    animation-duration: 2s;
    animation-delay: 0;
    animation-timing-function: ease;
    animation-iteration-count: 1;
    animation-fill-mode: none;
    animation-direction: normal;
    animation-play-state: paused;
}
```

Paused is a valid animation name. While it may seem that the animation named `paused` with a duration of 2s is being attached to the element or psuedo element, that is not what is happening. Because words within the shorthand animation are first checked against possible valid values of all animation properties other than `animation-name` first, therefore `paused` is being set as the value of the `animation-play-state` property.

```
#anotherFailedAnimation {
    animation: running 2s ease-in-out forwards;
}
```

The above is the equivalent to

```
#anotherFailedAnimation {
    animation-name: none;
    animation-duration: 2s;
    animation-delay: 0s;
    animation-timing-function: ease-in-out;
    animation-iteration-count: 1;
    animation-fill-mode: forwards;
    animation-direction: normal;
    animation-play-state: running;
}
```

Likely the developer has a keyframe animation called `running`. The browser, however, sees the term and assigns it to the `animation-play-state` property rather than the `animation-name` property. With no animation name declared, there is no animation attached to the element.

In light of this, `animation: 2s 3s 4s;` may seem valid, as if the following were being set:

```
#invalidName {
    animation-name: 4s;
    animation-duration: 2s;
    animation-delay: 3s;
}
```

If we remember from the keyframe identifier section above, 4s is *not* a valid identifier. Identifiers can not start with a digit unless escaped. For this animation to be valid, it would have had to be written as `animation: 2s 3s \34 s;`

To attach multiple animations to a single element or pseudo element, comma separate the animation declarations.

```
.snowflake {
  animation: 3s ease-in 200ms 32 forwards falling,
             1.5s linear 200ms 64 spinning;
}
```

Our snowflake with fall while spinning for 96 seconds, spinning twice during each 3s fall. At the end of the last animation cycle, the snowflake will stay fixed on the last keyframe of the falling @keyframes animation. We declared 6 of the 8 animation properties for the falling animation and 5 for the spinning animation, separating the two animations with a comma.

While you'll most often see the animation name as the first value as it's easier to read that way, because of the issue with animation property keywords being valid keyframe identifiers, it is not a best practice. That is why we put the animation name at the end.

It is fine, even a good idea, to use the animation shorthand. Just remember the placement of the duration, delay and name within that shorthand are important and omitted values default to their default values. Also, it is a good idea to not use any animation keyterms as your keyframe identifier.

Animation, Specificity and Precedence Order

In terms of specificity, the cascade, and which property values get applied to an element, animations supercede all other values and the cascade. When an animation is attached to an element, it takes precedence as if the specificity was even stronger than if the property values of that keyframe animation were set inline with an `!important`: `&lt;div style="keyframe properties here !important">`.

In general, the weight of a property attached with an ID selector 1-0-0 should take precedence over a property applied by an element selector 0-0-1. However, if that property value was changed via a keyframe animation, it will be applied as if that property/value pair were added as an inline style with an added !important.

A property added via a CSS animation, even if that animation was added on a CSS block that had very low specificity, will be applied to the element, in-spite of there being the same property applied to the same element via a more specific selector, or even an inline style, or even the keyterm !important on three nested id selectors. If an !important is declared on a property value within the cascade, that will not override the style that was added with an animation. (ex. http://codepen.io/estelle/pen/iDvBz). The animation is "even more !important".

That being said, don't include `!important` within your animation declaration block: the property/value upon which it is declared will be ignored.

If there are multiple animations specifying values for the same property, the property value from the last animation applied will override the previous animations.

```
#colorchange {
  animation-name: red, green, blue;
  animation-duration: 11s, 9s, 6s;
}
```

In the code example above, if red, green and blue are all keyframe animations that change the color property to their respective names, once the `animation-name` and `animation duration` properties are applied to #colorchange, for the first 6 seconds the property values in blue will be applied, then green for 3 seconds, then red for 2 seconds, before returning to its default property values. If `animation-fill-mode: both;` were added to the mix, the color would always be blue, as the last animation, or blue, overrides the previous green animation, which overrides the red first animation.

The default properties of an element are not impacted before the animation starts, and the properties return to their original values after the animation ends unless an `animation-fill-mode` value other than the default `none` has been set.

When an animation is attached to an element, the properties of animation keyframes only effect the element on which they are applied while the animation is iterating, which is the time after the `animation-delay` has expired, through the completion of the last animation iteration. Again, you can ensure the keyframe animation properties impact the element before the expiration of the `animation-delay` and after the last iteration ends with the `animation-fill-mode` property.

Animation iteration and display: none;

If the `display` property is set to `none` on an element, any animation iterating on that element or its descencdants will cease, as if the animation were detached from the element. Updating the `display` property back to a visible value will reattach all the animation properties, restarting the animation from scratch.

```
.snowflake {
  animation: spin 2s linear 5s 20;
}
```

The snowflake will spin 20 times, each spin takes 2 seconds, with the first spin starting after 5 seconds. If the snowflake element's `display` property gets set to none after 15 seconds, it would have completed 5 spins before disappearing (5 second delay, then 5 spins at 2 seconds each). If the snowflake `display` property changes, making it visible again, a 5 second delay will elapse before it starts spinning 20 times. It makes no differnce how many animation cycles iterated before it disappeared from view the first time.

Animation and the UI Thread

Note that CSS animations have the lowest priority on the UI thread. If you attach multiple animations on page load with positive values for `animation-delay`, the delays expire as prescribed, but the animations will not begin until the UI thread is available to animate. For example, if you have 20 animations set with an animation delays to start animating at one second intervals over 20 seconds, with their `animation-delay` properties set to 1s, 2s, 3s, 4s and so on, if the document or application takes a long time to load, with 11 seconds between the time the animated elements were loaded and the UI thread finished drawing the page, and downloading, parsing and executing any JavaScript, the animation delays of the first 11 animations will have expired, and will all commence when the UI thread has become available. The remaining animations will each then begin animating at one second intervals.

> While you can use animations to create changing content, dynamically changing content can lead to seizures in some users. Always keep accessibility in mind, ensuring the accessibility of your website to people with epilepsy and other seizure disorders.

Animation Events

There are a few animation-related events you can access with DOM event listeners.

`animationstart`

The `animationstart` event occurs at the start of the animation. If there is an `animation-delay` then this event will fire once the delay period has expired. If there is no delay, `animationstart` event occurs when the animation is applied to the element. If there are no iterations, the `animationstart` event still occurs. If there are multiple animations attached to an element, an `animationstart` event will occur for each of the applied valid keyframe animations.

`animationend`

The `animationend` event occurs at the conclusion of the last animation. It only occurs once per applied animation: if an element has 3 animations applied to it, the `animationend` event will occur 3 times: at the end of the last iteration, which is usually equivalent to the result of the following equation:

(animation-duration * animation-iteration-count) + animation delay = time

If there are no iterations, the `animationend` event still occurs once for each animation applied. If the `animation-iteration-count` is set to `infinite`, the `animationend` event never occurs.

`animationiteration`

The `animationiteration` event occurs at the end of each iteration of an animation, before the start of the next iteration. If there are no iterations, or the iteration count is less than or equal to one, the `animationiteration` event never occurs. If the iteration count is infinite, the `animationiteration` event occurs ad infinidum. Unlike the `animationstart` and `animationend` events with each occur once for each animation name, the `animationiteration` event can occur multiple times or no times per animation name, depending on how many iterations occur. Note the event happens between animation cycles, and will not occur at the same time as an `animationend` event.

Printing Animations

While not actually "animating" on a printed piece of paper, when an animated element is printed, the relevant property values will be printed. Obviously you can't see the element animating on a piece of paper, but if the animation caused an element to have a border-radius of 50%, the printed element will have a border-radius of 50%.