MOVE THINGS WITH CSS

JHEY TOMPKINS

https://jhey.dev

CONTENTS

HOWDY!

You're interested in making things move on your websites and in your apps with CSS. I'm glad you've chosen me to guide you. Thank you!

This book assumes you've never created a CSS animation or transition before. But, even if you have, there may be things you were not aware of that we'll cover here. This book does assume some familiarity with HTML and CSS.

This book covers both CSS animations and CSS transitions. Although the two are often paired together, they are separate topics. You don't need to read one before the other. If you want, you could skip right to animations after reading the background sections.

For CSS transitions, we walk through creating a button with micro-interactions. For CSS animations, we go through various demos to address the different properties.

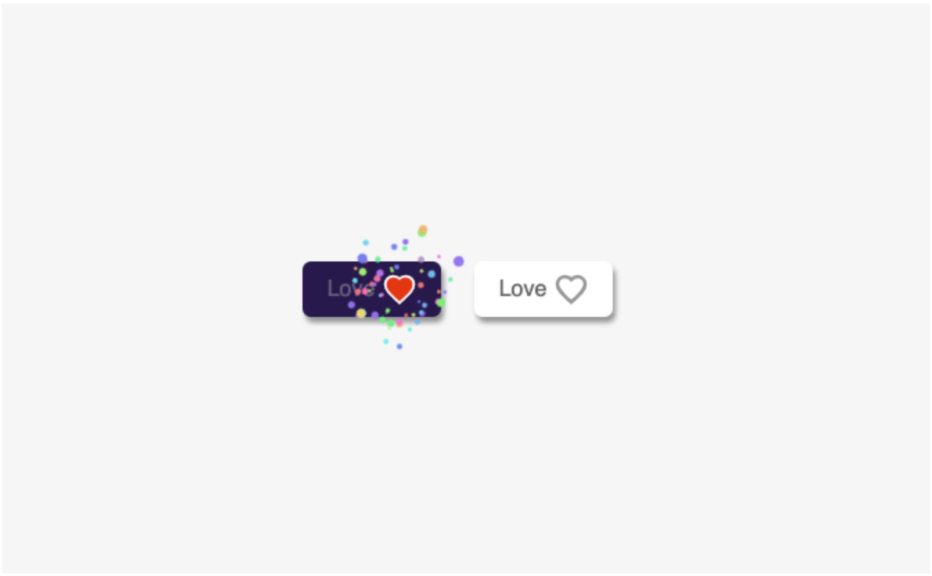
This static format isn't the best for showing animations. Look out for demo links throughout that will take you to interactive CodePen demos. The source code for every animation covered in this book is available to you in a CodePen collection. This means you can fork them, download them, edit them, etc. The code is also available from the Zip file or Github repository.

Look out for these callouts throughout the book. These might contain small challenges, questions, etc.

I'd love to know what you think of the content or see what you make with it. Share things with me and others over on Twitter using **#MoveThingsWithCSS**.

WHY MOVE THINGS?

The main reason? To enhance the usability and user experience for users. That doesn't mean we should use animation everywhere. There's a balance. The animation should enhance the user experience for a user, not distract or block them. Our aim is to direct, guide, and inform our users. Not deter them. Good use cases for animation include things like loading spinners and micro interactions. Poor examples? Those fancy hero animations that block users until they've finished spring to mind.



A micro-interaction button. Check the demo.

MOTION DESIGN

This is by no means a book on designing animation. But, we should mention it. Implementing animations and transitions is only one part of the puzzle. Making our animations feel natural and non-intrusive is very important. Good animation should almost go unnoticed. It enhances the user experience without becoming a talking point.

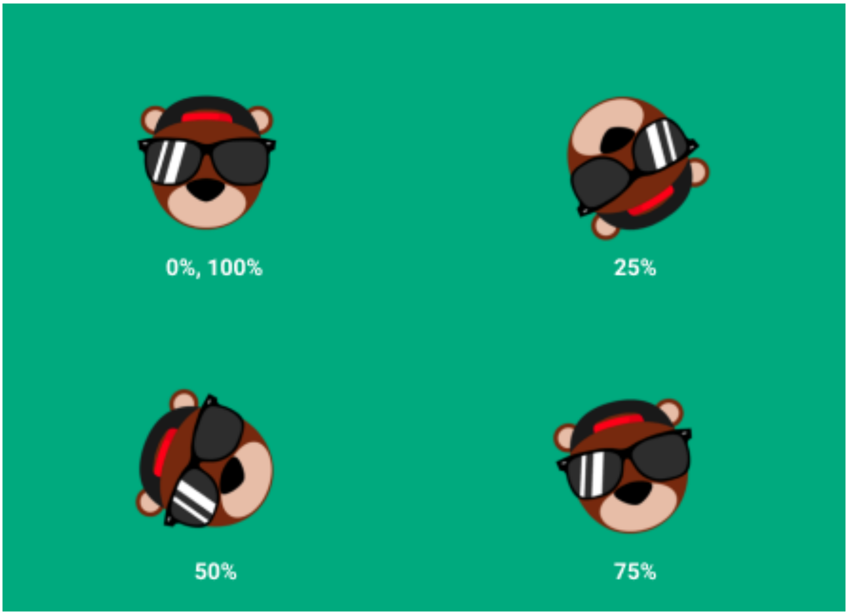
Fading in a list of items.

Luckily for us, we have a very good set of principles we can follow for designing motion. These are Disney's “Twelve basic principles of animation”, first introduced in 1981. They still hold true now and following them is a good move. This article does a great job of showing ways in which to apply them to modern interface design. Aside from those principles, “The ultimate guide to proper use of animation in UX” is a must read. It’s full of great animated examples and golden tips.

It's worth noting that many of the animations we cover will be basic implementations. The examples could be great candidates for integrating some of those principles.

OUR FIRST ANIMATION

Let's hop right in and create an animation. For this animation, we will make an element spin 360 degrees.



The timeline for our first animation. Spinning the bear!

First, we create our animation using the `@keyframes` rule. The `@keyframes` rule takes the following structure.

```
@keyframes [ NAME ] {
  [ KEYFRAME SELECTOR ] { CSS STYLES }
}
```

`NAME` is the name we give to our animation. You can have one or many keyframe selectors. We will name our animation “spin”. To spin our element we can use the `transform` property and rotate from `0deg` to `360deg`. We use two keyframe selectors. One to define the start of our animation(`from`) and one for the end of our animation(`to`). The `from` and `to` keywords are equal to `0%` and `100%`.

```
@keyframes spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

We can take this further. The `from` styles match the initial `transform` of our element. Thus, the keyframe selector is redundant. We can remove it.

```
@keyframes spin {
  to {
    transform: rotate(360deg);
  }
}
```

Now, we need to apply that animation to our element. We can use the `animation-name` and `animation-duration` properties.

```css
.bear {
  animation-duration: 2s;
  animation-name: spin;
}
```

Here we are telling our element to use the animation `spin` with a duration of 2 seconds.

> Durations for animations and transitions can be set in either seconds(s) or milliseconds(ms) **#MoveThingsWithCSS**

We're only a few pages in and from that first animation, we have enough to go off and start creating cool animations. Check out the demo. We will dig into animations further later on. Before we do that, let's take a look at a few other things.

WHAT CAN WE ANIMATE?

We've made our first animation. We've got a taste for it. Now we're thinking "What are the possibilities?". Before we deep dive it's important to gain an understanding of what things we can animate. This list from MDN contains all animatable CSS properties. Lea Verou also put together a great demo page for checking out animatable properties.

We can't animate every CSS property. A good example being the `display` property. We may think to animate that to show and hide an element. And although it will run in a browser. We don't see anything.

Because we can, doesn't mean we should. Of all the animatable properties, we may choose to animate some over others. This is for performance reasons. In ideal situations, we want a machine's GPU to handle the heavy lifting when animating a property. The animation of some properties can trigger layout recalculations or several repaints.

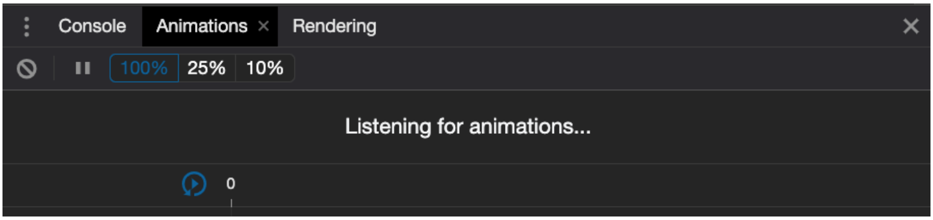
A good rule is to try and stick to what your browser can animate for little performance cost. That's either `transform` or `opacity`. For example, if we want to move an element, it's more performant to use `transform: translate(x, y)`. An example of a less performant animation may be animating the `width` and `height` of an element. This could trigger layout changes for other elements and becomes costly.

For more on animation performance, this article on html5rocks does a great job of digging in. There is also csstriggers.com(It's a PWA too). It's a handy reference for checking the cost of animating various CSS properties.

ANIMATION INSPECTOR

When creating animations, especially complex animations, we need a way to debug them. This is where Google Chrome's Devtools Animation Inspector comes into play.

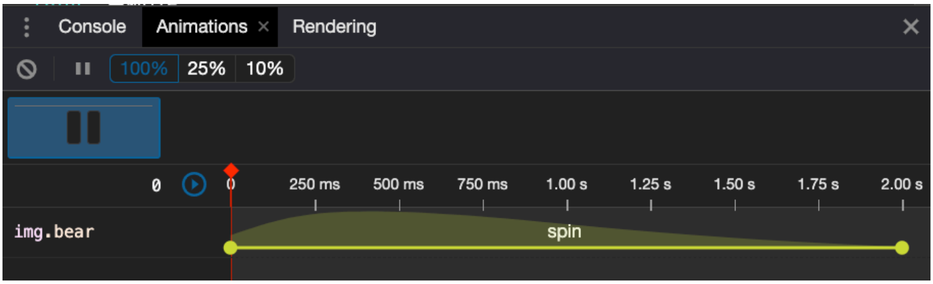
Open a page with that has CSS animation on it (You could use the demo for our first animation) in Google Chrome. Open the Developer Tools. Open up the Animations panel by going into “More tools”. If the Animations panel says "Listening for animations...", refresh the page.



Listening for animations...

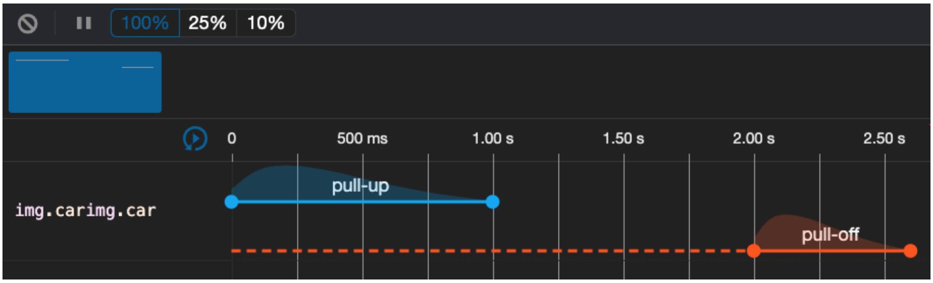
Animations panel listening for animations.

After a page refresh, we should see something in the “Animations” panel. That's the animation!



Our “spin” animation shows in the panel.

Click the animation and we are able to inspect it. This particular animation isn't complex. The Animation Inspector allows us to do various things though. We can experiment with durations and delays as well as altering playback speeds. It's great for visualizing the composition of animations. We can also replay animations without having to refresh the page. Dragging the indicator will allow you to scrub animations.



More complex animation timeline showing composition.

I recommend using the Animation Inspector for the various demos. It will give a better insight into how useful it can be.

COMMON VALUES AND OVERLAP

As we're covering both animations and transitions, there's going to be some overlap. Let’s cover that before we dig in.

COMMA SEPARATED VALUES

The animation and transition properties can accept many comma-separated values. This feature allows us to define many animations for an element in a single declaration. For example, we could transition two properties for an element with different speeds. Or we may apply many animations to an element at the same time. It opens up many possibilities. Consider our first animation. We could make the bear vanish in the middle.

```css
.bear {
  animation-duration: 2s;
  animation-name: spin, vanish;
}
```

Check the demo. Now two animations run on the element at the same time with the same duration. If we wanted to run two animations, one after the other, we'd need to use delays. More on this later.

Read more about setting many animation values in this section on MDN.

DURATION

We touched on durations when making our first animation. Much like we can define `animation-duration`, we can define `transition-duration` too. Durations can be set in either millisecond(ms) or second(s). We also use these time values for `animation-delay` and `transition-delay`.

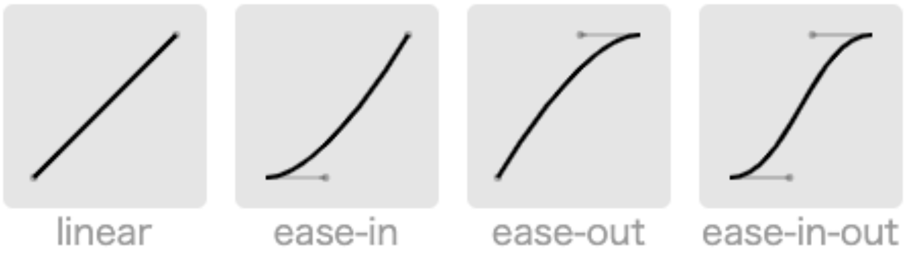
TIMING FUNCTION

The timing function defines how a value changes. By this, we mean the speed at which a value changes over the duration of an animation or transition. The `animation-timing-function` and `transition-timing-function` properties define speed characteristics. We often refer to these as the “ease" for animation or transition. Think of the easing values

We have some easing options out of the box:

- `ease`(default) - start and end slowly but speed up in the middle.
- `ease-in` - start slowly
- `ease-in-out` - start and end slowly
- `ease-out` - end slowly
- `linear` - maintain speed throughout
- `steps(number, direction <optional>)` - provides a way to split the animation into equal steps. The direction value can either be start or end. `start` means that the first step happens at the start of the animation. `end` means that the last step happens at the end of the animation. `end` is the default direction.
- `cubic-bezier(x1, y1, x2, y2)` - provides a means to define custom speed characteristics based on a bezier curve.

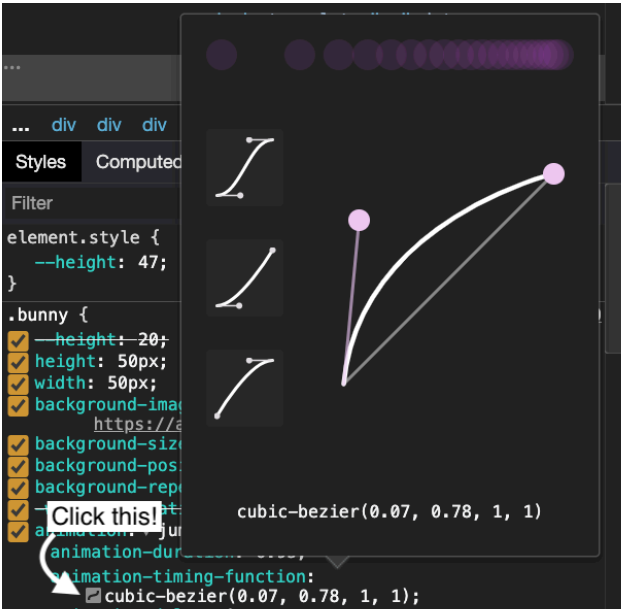
Of the various easing options, `cubic-bezier` is likely the trickiest to grasp. It defines a cubic bezier curve that maps to the speed of your animation or transition. But, that curve is how we can visualize any ease, not including `steps`. Aside from `steps`, the other ease values are shorthand for `cubic-bezier` configurations. For example, `linear` ease would be equal to `cubic-bezier(0, 0, 1, 1)`.



Common eases visualized courtesy of cubic-bezier.com

This site is great for visualizing `cubic-bezier` equivalents of the other easing values. You can also configure the speed graph and experiment with different speeds. You might also be interested in easings.net.

Depending on which browser you are using, your browsers' dev tools can also help. For example, Google Chrome's Devtools has a great ease visualizer. You can switch between various pre-configured eases. Or you can drag the handles around to create your own custom ease.



```
element.style {
  --height: 47;
}
.bunny {
  --height: 20;
  height: 50px;
  width: 50px;
  background-imag
        https://a
  background-siz
  background-pos
  background-rep
  animation-dur
  animation-timing-function:
    cubic-bezier(0.07, 0.78, 1, 1);
```

```
cubic-bezier(0.07, 0.78, 1, 1)
```

Google Chrome's easing visualiser in action.

Which ease do you choose? Different scenarios call for different easing. This article is a nice intro to the basics of easing. It's also a good time to refer to the principles of animation we mentioned earlier. Experiment with different easing to find what feels right for your application. For example, a lightbulb switching on and off might use `steps(1)`. But, a rocket ship launching could use `ease-in`.

Check out this demo that visualises how the different eases can affect the same animation **#MoveThingsWithCSS**

CSS TRANSITIONS

Often, we can achieve our goals with CSS transitions alone. We may not need to reach for CSS animations.

CSS transitions allow us to define the speed and duration at which a property value changes. The transitioning of a value change can make things easier on the user's eyes. Whereas, sudden dramatic changes can be jarring or even more distracting to a user. Common use cases for transitions include micro-interactions and UI state changes. But above all else, transitions can be that little detail that makes your UI/UX stand out.

AN EXAMPLE

To showcase CSS transitions, let's create a button that has some micro-interaction. This button will be a "Bookmark" button. Clicking the button toggles the visual state of it. The little star fills in when the toggle is active.

Bookmark
Bookmark

Our micro-interaction button where a star fills.

Here's the demo for our button in a transition-less state. Clicking the button fills the star. Note how the different states affect smaller details. When we hold our cursor down, the shadow of the button disappears.

TRIGGERING

It's worth noting how we can trigger transitions. In our example currently, we are toggling a class on the `button` element. That gives us an opportunity to change a property value for an element. But, we don't need to rely on class changes only. Many of the CSS pseudo-classes are great opportunities to trigger a transition. For our button, `:hover` and `:active` our great options. Applying different transitions during different states can be very effective. We will look at an example of how to do this.

OUR FIRST TRANSITION

Let's add some transitions to our button. We've got two opportunities here. We can add transitions to the actual button and the star icon within it.

Let's start with the `button` element. If we look at the CSS for our button, it changes `background-color` on `:hover`, and `box-shadow` on `:active`.

```css
.bookmark-button:hover {
  background-color: hsl(280, 50%, 0%);
}

.bookmark-button:active {
  box-shadow: 0 0 0 0 hsl(0, 0%, 0%);
}
```

Let's transition the `box-shadow`. To do this we can use the `transition` property.

```css
.bookmark-button {
  transition: box-shadow 0.2s;
}
```

That's it! Check the updated demo. Exaggerate your cursor press to appreciate the `transition`. Now the `box-shadow` transitions and it makes the button press feel more "real". Compare the updated demo to the original. It's a very subtle change.

Bookmark
Bookmark
Bookmark

The stages of our button press. Default, pressed, and active.

STRUCTURE

What did we do there? It wasn't overly exciting, was it? Don't worry, we'll spice it up. We told the browser to transition any change in the `box-shadow` property over a duration of `0.2` seconds. The `transition` property is a shorthand for defining the four transition properties.

- `transition-property` - defines the name or names of properties to transition. The keyword `all` will transition all properties and is the default value.
- `transition-duration` - defines the duration of a transition in millisecond(ms) or seconds(s).
- `transition-timing-function` - defines the ease of a transition using the timing functions we looked at earlier.
- `transition-delay` - defines the delay for a transition in either millisecond(ms) or seconds(s).

Remember that all these properties accept more than one value, comma-separated **#MoveThingsWithCSS**

In our first transition, we used the shorthand notation with the default `transition-timing-function` and `transition-delay`. That would've been equal to writing.

```css
.bookmark-button {
  transition-property: box-shadow;
  transition-duration: 0.2s;
  transition-timing-function: ease;
  transition-delay: 0s;
}
```

Now we've got that out of the way, let's have a little fun with it!

Want to have some fun?

```css
* { transition: all 0.3s; }
```

Add that to any page's CSS. Now resize the page or interact with the content! **#MoveThingsWithCSS**

TRANSITION-PROPERTY

The `transition-property` property allows us to define the property whose value should transition. We're transitioning the `box-shadow` for our button. Let's also transition the `background-color`.

```
.bookmark-button {
  transition-duration: 0.2s;
  transition-property: background-color,
                       box-shadow;
}
```

How else might we write that? Could we chain the shorthand? **#MoveThingsWithCSS**

And lastly, let's transition the `fill` of the star.

```
.bookmark-button__star {
  transition: fill 0.2s;
}
```

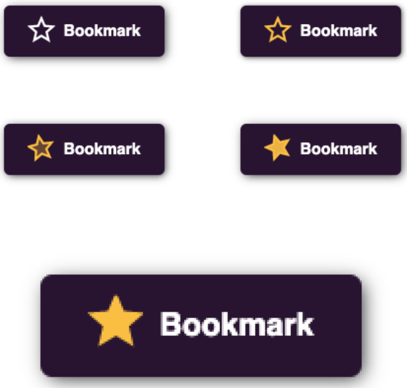
It's still lacking something if we look at the demo. The transition is a nice touch. It's subtle, which is good. Some of the best animations are those that go almost unnoticed.

But we want something to pop here. Let's spin the star round when the button becomes active. That'll give us something we can actually show off our transition skills with.

```
.bookmark-button__star {
  transition: fill 0.2s, transform 0.2s;
}

.bookmark-button--active
.bookmark-button__star {
  transform: rotate(216deg);
}
```

Now the star spins when the button is pressed and it gives our button something extra. Check the demo. The button element transitions are subtle and we can leave them as is. But, there is room to make the star transition more appealing.



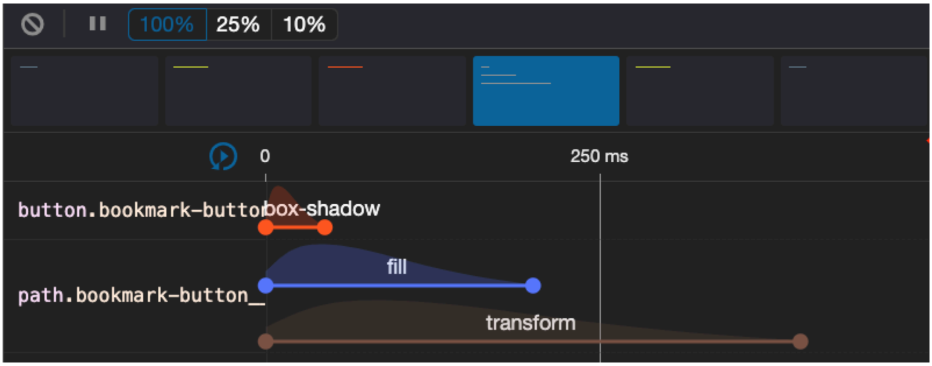
The star spins and changes fill over the transition period.

TRANSITION-DURATION

The `transition-duration` property allows us to define the duration it takes for a property value to change. Remember, we can define this in millisecond(ms) or seconds(s). We may decide that the star changes color too slow and spins too fast. We can adjust the `transition-duration` for each property. Here we keep the `fill` transition and slow down the `transform` transition.

```css
.bookmark-button__star {
  transition-property: fill, transform;
  transition-duration: 0.2s, 0.4s;
}
```

Checking the demo, we now have a slower spin.



Confirming the transition timeline in the Animation Inspector.

TRANSITION-TIMING-FUNCTION

The `transition-timing-function` property allows us to define the speed characteristics for a transition. Let's add a little bounce to the spin of our star. For this, we could use a custom speed via `cubic-bezier`.

```
.bookmark-button__star {
  transition-property: fill, transform;
  transition-duration: 0.2s, 0.6s;
  transition-timing-function: ease,
                     cubic-bezier(.5,-1.5,.6,2.5);
}
```

Here, we adjust the `transition-timing-function` for the `transform` transition of the star. To appreciate the effect better, we bump the `transition-duration` for `transform`. Check the demo.

Remember, we can use the Animation Inspector to inspect transitions as well as animations **#MoveThingsWithCSS**

TRANSITION-DELAY

The `transition-delay` property allows us to delay a transition. What if we delayed the fill of our star until it has almost spun?

```
.bookmark-button__star {
  transition-property: fill, transform;
  transition-duration: 0.2s, 0.6s;
  transition-timing-function: ease,
                    cubic-bezier(.5,-1.5,.6,2.5);
  transition-delay: 0.4s, 0s;
}
```

Delaying the fill of the star gives a nice staged effect as it spins. Check the demo. We could even chain the transitions and have the fill happen after the spin. We'd do this with:

```
.bookmark-button__star {
  transition-delay: 0.6s, 0s;
}
```

EXTRA TOUCHES

Almost there. There's a couple of improvements we could make. When we hover our button, there's no sign that the star will do anything. We could rotate it when we hover our button.

```
.bookmark-button:hover .bookmark-button__star {
  transform: rotate(45deg);
}

.bookmark-button--active:hover
.bookmark-button__star,
.bookmark-button--active
.bookmark-button__star {
  transform: rotate(216deg);
}
```

Note how we need to make sure we cover the `:hover` state when the button is active too now. This is so our star still rotates to `216deg` when we are hovering it.

Oh no. The timing function for the star looks off when we aren't fully rotating it. This leads us to how we can apply different transitions for different states of our UI. For our button, we may decide to only use our custom speed when the button is active.

```
.bookmark-button__star {
  transform-origin: 50% 50%;
  transition-property: fill, transform;
  transition-duration: 0.2s;
}

.bookmark-button--active
.bookmark-button__star {
  transition-duration: 0.2s, 0.6s;
  transition-timing-function: ease,
                    cubic-bezier(.5,-1.5,.6,2.5);
}
```

And that's it. That's our micro-interaction button using CSS transitions. Check out the demo.

What else could we transition or change to make our button even better? **#MoveThingsWithCSS**

CSS ANIMATIONS

We made our first animation earlier. Let's dig further into CSS animations!

ANIMATION-NAME

Covered in the first animation we made. The `animation-name` property specifies the name of the `@keyframes` our element should use. You can also pass the keyword `none` so that there is no animation.

ANIMATION-DURATION

Covered in the first animation we made. The `animation-duration` property specifies how long an animation should play for. We can define durations in either millisecond(ms) or seconds(s).

@KEYFRAMES

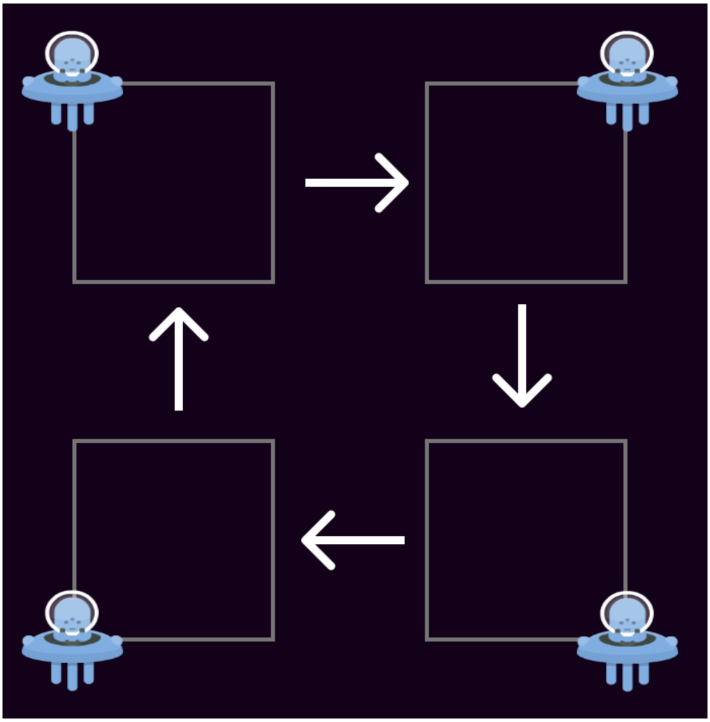
We put together our first `@keyframes` rule in our `spin` animation.

```
@keyframes spin {
  to {
    transform: rotate(360deg);
  }
}
```

After specifying an animation name, we structure our animation within keyframe selectors. The keyframe selector specifies a percentage of the animation duration. Or, as mentioned before, we can use the `from` and `to` keywords that are equal to `0%` and `100%`.

Each selector defines styles that should apply at that point in the animation. If we have selectors that specify the same CSS styles, we can group them together.

Let's start with a basic example. Consider the animation of a little spacecraft moving around the path of a square.



The stages of our animation.

We will call our animation `squarePath`. For this example, there will be four positions for our element. For every side of the square, we use a quarter of the animation.

Because our start and finish position will be the same, we can group those keyframe selectors.

```
@keyframes squarePath {
  0%, 100% {
    transform: translate(-100%, -100%);
  }
  25% {
    transform: translate(100%, -100%);
  }
  50% {
    transform: translate(100%, 100%);
  }
  75% {
    transform: translate(-100%, 100%);
  }
}
```

Apply the `animation-name` and `animation-duration` to our element.

```
.spacecraft {
  animation-duration: 2s;
  animation-name: squarePath;
}
```

And that's it! We have a little spacecraft moving along the path of a square. Check out the demo.

ANIMATION-ITERATION-COUNT

If we want an animation to run more than once, we can use `animation-iteration-count`. The property accepts a number or the keyword `infinite`.

In our example, the spacecraft travels around the square and then returns to the center. It's quite sudden and a little jarring. What if we want the spaceship to keep traveling on that path? If we want an animation to run an infinite amount of times, we use the keyword `infinite`.

```css
.spacecraft {
  animation-duration: 2s;
  animation-name: squarePath;
  animation-iteration-count: infinite;
}
```

Now our spaceship moves along the path an infinite amount of times. Check the demo.

Could you add some stars or something extra to the scene? How about removing the square? **#MoveThingsWithCSS**

Consider another demo where we have a racecar going round a track. It might be the case that this race has 200 laps.

A racer going round a track.

An `animation-iteration-count` of 200 makes our racecar do 200 laps before coming to a stop.

```
.racecar {
  animation-name: lap;
  animation-duration: 2s;
  animation-iteration-count: 200;
}

@keyframes lap {
  to {
    transform: rotate(360deg);
  }
}
```

Check out the demo. I wouldn't recommend watching all 200 laps!

Could you add a second car and make it a race?
#MoveThingsWithCSS

ANIMATION-TIMING-FUNCTION

The `animation-timing-function` allows us to define the speed of an animation. We discussed eases and timing functions earlier.

Let's say we decide to make the car do an `infinite` amount of laps. It could be a loading animation. Something looks off. There's a little break after every lap. To fix this we can adjust the timing function of our animation. For our racecar, a `linear` timing function will work well.

```css
.racecar {
  animation-name: lap;
  animation-duration: 2s;
  animation-iteration-count: infinite;
  animation-timing-function: linear;
}
```

Now checking the updated demo, the car races around the track at a constant speed with no breaks.

Now we have the tools to make a loading animation. Could you make one and share it? **#MoveThingsWithCSS**

ANIMATION-PLAY-STATE

Use the `animation-play-state` property to change the play state of our animations. Our animations can either run or we can pause them. To do this we use the `animation-play-state` property which defaults to `running`.

For our racecar demo, we could introduce a checkbox that toggles the play state and stops the racecar. Like a flag! When it's `:checked` we set the `animation-play-state` of the racecar to `paused`. For this, we use the sibling combinator.

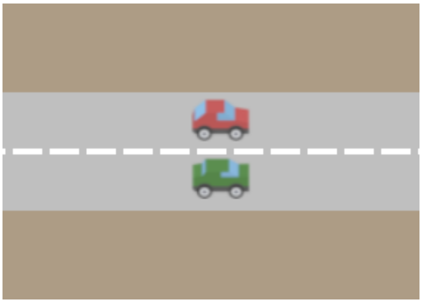
```css
:checked ~ .racetrack .racecar {
  animation-play-state: paused;
}
```

Check the demo and try out the checkbox. Here's another demo where we can toggle a cellphone ringing by tapping the page. Use your browser dev tools to inspect and experiment with the different animations. The demo uses the pseudo-elements belonging to the `.cellphone` element.

Ringing cellphone demo.
Tap anywhere to toggle ringing.

ANIMATION-DELAY

We can delay an element's animation using `animation-delay`. Delaying animations gives us the opportunity to stagger animations. This could be particularly useful when rendering lists of elements for example. There are some good examples of this in the motion design articles we referenced earlier.



A very boring drag race.

Let's start with a basic example. Let me introduce you to the most boring drag race. Two cars race along the strip at the same speed and take the same time.

```
.car {
  animation: race 4s;
  transform: translate(calc(-40vw + 25px), 0);
}
@keyframes race {
  to {
    transform: translate(calc(40vw -25px), 0);
  }
}
```

Now, it's unlikely the drivers have the exact same reaction time. Let's make one slower than the other.

```css
.car--green {
  animation-delay: 0.25s;
}
```

Now if we check the demo, the red car pulls off quicker than the green one.

> Could you make it so that green car finishes before the red car without changing the delay? **#MoveThingsWithCSS**

We aren't quite done with `animation-delay` yet. It's important to highlight a couple of things to be mindful of with `animation-delay`.

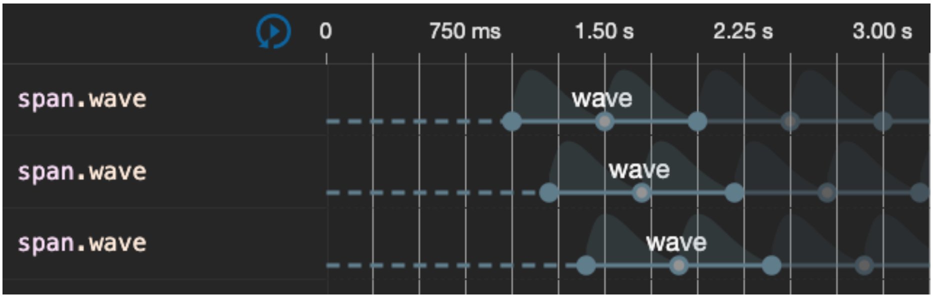
Consider this demo where all the hands raise at the same time.

Three sets of raised hands in unison.

If we increment the `animation-delay` for each set of hands, we can create a basic “Wave”. Check the demo.

```
.wave:nth-of-type(1) {
  animation-delay: 1s;
}
.wave:nth-of-type(2) {
  animation-delay: 1.2s;
}
.wave:nth-of-type(3) {
  animation-delay: 1.4s;
}
```

That looks great. But what if we want the wave to show an infinite amount of times? Let's make the `animation-iteration-count` infinite and check the demo. Hold up. Did we lose the delay? Correct! The `animation-delay` is only applied once before the animation starts. It doesn't apply to every iteration of an animation. We've maintained the stagger, but lost the delay. We could experiment with different delays or reach for a JavaScript solution. But one neat trick is to adjust the `@keyframes` so the delay is considered.



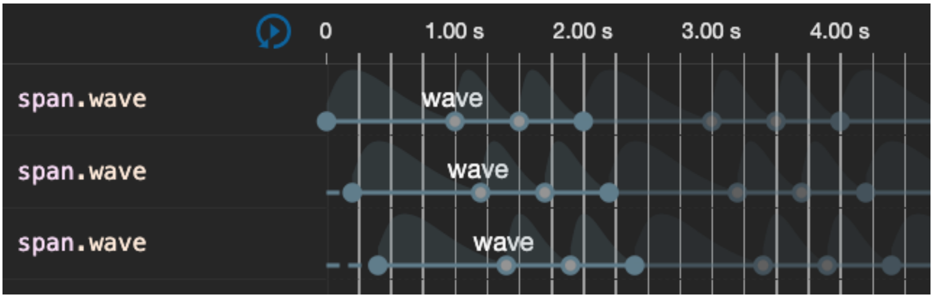
We can see the initial delay in the Animation Inspector but it doesn't persist.

If we look at our delays, we could subtract `1s` and make that part of the keyframes. To integrate the delay we could then increase the duration by that `1s` and pad the start of the animation.

We update our code and pad the start of the animation with nothing:

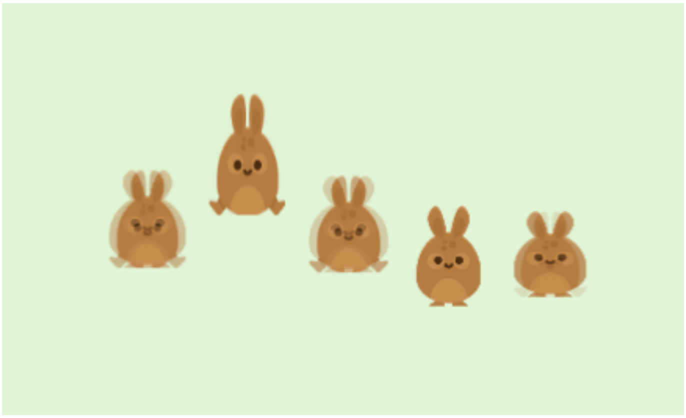
```
.wave {
  animation: wave 2s infinite;
}
.wave:nth-of-type(1) {
  animation-delay: 0s;
}
.wave:nth-of-type(2) {
  animation-delay: 0.2s;
}
.wave:nth-of-type(3) {
  animation-delay: 0.4s;
}
@keyframes wave {
  0%, 50%, 100% {
    transform: translate(0, 0);
  }
  75% {
    transform: translate(0, -50%);
  }
}
```

Where before we were using the `0%` to `100%` space, `50%` to `100%` is now the space for our animation. The wave needs to raise now at `75%`. If we updated `50%` to `75%` and left it, we wouldn't get the delay. Setting the translate at `0%`, `50%`, and `100%` ensures the initial state during the delay period. Check the demo and we have a persistent delay for each wave.



Now our animation takes into account the integrated delay.

For some animations, we may want to skip the start of a timeline. Using a negative delay allows us to skip the duration defined as if it had already played. For example, if we're animating a scene with many animations, it might look odd if they all stagger in. This is particularly useful where we want to animate a group of things with a staggered delay. Consider the bunnies in this demo.



Jumping bunnies!

If you watch the demo, there's an awkward section at the start before they all start jumping. To get rid of this, switch the animation delays to negative.

```
.bunny:nth-of-type(1) {
  animation-delay: 0s;
}
.bunny:nth-of-type(2) {
  animation-delay: -0.2s;
}
.bunny:nth-of-type(3) {
  animation-delay: -0.4s;
}
.bunny:nth-of-type(4) {
  animation-delay: -0.6s;
}
.bunny:nth-of-type(5) {
  animation-delay: -0.8s;
}
```

Now check the demo. The awkward section is gone and those jumpers keep on jumping.

> Negative delays can be quite powerful. Consider this demo. It uses negative **animation-delay** to create a DIY animation scrubber. It's also a visual way to understand negative **animation-delay #MoveThingsWithCSS**

We've been using `:nth-of-type` for specifying specific delays, durations, etc. One tip here is to make use of CSS variables. It makes maintaining stagger effects, etc. easier. We could refactor our bunny code to use CSS variables.

```css
.bunny {
  animation-delay: calc(var(--index) * -0.2s));
}
.bunny:nth-of-type(1) {
  --index: 0;
}
.bunny:nth-of-type(2) {
  --index: 1;
}
.bunny:nth-of-type(3) {
  --index: 2;
}
.bunny:nth-of-type(4) {
  --index: 3;
}
.bunny:nth-of-type(5) {
  --index: 4;
}
```

Now to adjust the stagger, we only need to change one rule where we define the delay.

In fact, leveraging CSS variables, we can create many interesting staggered variations. Check out this demo and move the slider. Can you create any other variations? Share it! **#MoveThingsWithCSS**

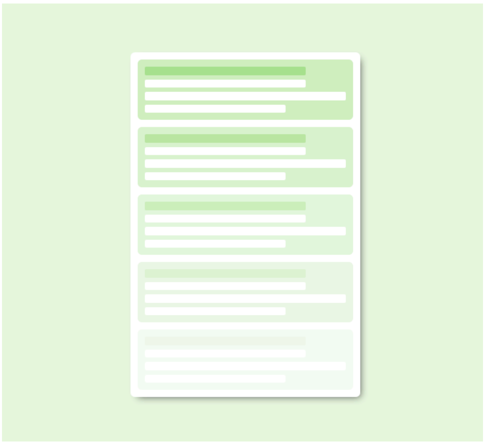
ANIMATION-FILL-MODE

This one's magic. The `animation-fill-mode` property defines how animation affects an element before and after animation. By default, when we apply an animation to an element, it has no effect before or after it has run.

The values we can use for `animation-fill-mode`:

- `none` - element retains no styling.
- `forwards` - element retains animation styling from final keyframe.
- `backwards` - element retains styling from first keyframe.
- `both` - element retains styling in both direction.

Remember, when discussing animation-delay, we said how animating a list would make a good stagger? Consider this demo where a list of elements fades in with a stagger.



List items that fade in one after another.

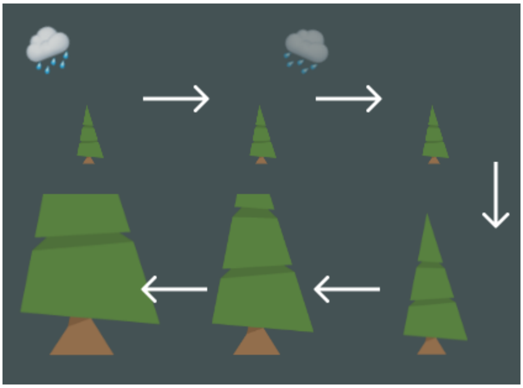
It doesn't look right at all. By default, each element has an `opacity` of `1` so they are visible until the animation starts. We want every element to be invisible. We can use `backwards`.

```css
.contact {
  animation-fill-mode: backwards;
}
```

Check the demo. Now the list elements fade in and remain as we would expect.

How might we change the demo to use **animation-fill-mode: forwards** but work exactly the same? **#MoveThingsWithCSS**

The need for `animation-fill-mode` becomes more clear when we start creating timelines. Consider this demo where a rain cloud animates before a tree grows. Check out the `animation-fill-mode` declarations.



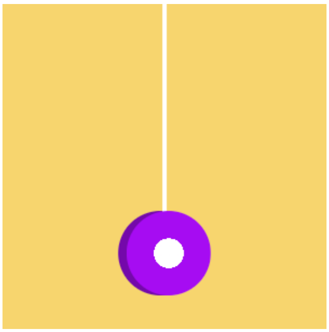
Tree growing demo with fill mode.

ANIMATION-DIRECTION

The `animation-direction` property allows us to define the direction of the animation. There are four values of which two take into consideration if your animation runs more than once.

- `normal` - animation plays as defined.
- `reverse` - animation plays in reverse.
- `alternate` - the direction of the animation alternates on each play.
- `alternate-reverse` - the direction of the animation alternates on each play but starts in reverse.

How might we use `animation-direction`? We could use it to communicate undoing something. Or we may have something that opens and closes. It's quite handy for yoyo effects. Let's make a yoyo.



Our yoyo.

If you check the demo, the yoyo drops in and does nothing after that. To make it go back up, we could use the `alternate` direction value with an `infinite` iteration count.

```
.yoyo {
  animation-iteration-count: infinite;
  animation-direction: alternate;
}
```

Now check the demo. Our yoyo goes up and down and we only define a keyframe that takes it in one direction.

How else could we make this animation? Can you make it so that the yoyo hangs at the bottom for a bit before going back up? **#MoveThingsWithCSS**

ANIMATION

We've covered all the specific animation properties. The last one is the `animation` property. It's a shorthand property for defining animations.

If we had the following animation code:

```css
.plane {
  animation-delay: 1s;
  animation-direction: normal;
  animation-duration: 1s;
  animation-fill-mode: backwards;
  animation-iteration-count: 1;
  animation-name: fly-in;
  animation-play-state: running;
  animation-timing-function: ease;
}
```

It would be equal to writing this:

```css
.plane {
  animation: fly-in 1s 1s ease 1 normal
             backwards running;
}
```

Ordering of properties is important. The most important being durations. The first duration specified will apply to `animation-duration`. If there's a second, it will apply to `animation-delay`. This MDN documentation explains the syntax in greater detail.

Demo for little plane flying along.

If we make use of comma-separated values, we can apply many animations to an element in a more concise way. Consider this demo of a plane flying. Instead of:

```
.plane {
  animation: fly-in 1s 1s backwards,
             advance 4s 2s infinite;
}
```

The longhand version would be written as:

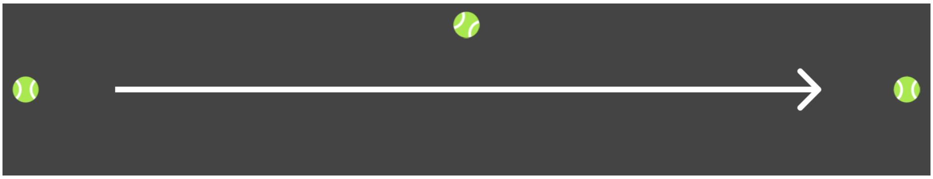
```css
.plane {
  animation-name: fly-in, advance;
  animation-duration: 1s, 4s;
  animation-delay: 1s 2s;
  animation-fill-mode: backwards, none;
  animation-iteration-count: 1, infinite;
}
```

Both versions work the same and it comes down to judgment or preference for which you use for your work. Sometimes the longhand version feels right for more complex scenarios where verbosity helps.

OTHER STUFF

We've covered CSS animations and transitions. You should now know enough to go and make some awesome things with movement. This section covers some interesting things that don't quite fit elsewhere.

CURVED ANIMATION PATHS


A ball being thrown from left to right.

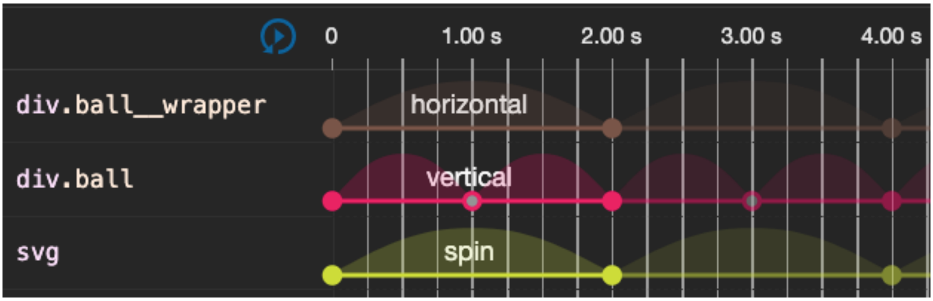
We can animate an element from point to point but how do we add a curve to our animation path? Let's consider an example where a ball is being thrown from one point to another.

```
.ball {
  animation: throw 2s infinite alternate
             ease-in-out;
}
@keyframes throw {
  0% {
    transform: translate(-25vw, 0);
  }
  50% {
    transform: translate(0, -20vh);
  }
  100% {
    transform: translate(25vw, 0);
  }
}
```

That code looks right. But check the demo. It's awkward. How do we get the desired effect? A wrapper element. If we wrap the ball with an extra element, we can use that to handle the horizontal movement. Then the ball only needs to worry about the vertical movement.

```
.ball {
  animation: vertical 2s infinite ease-in-out
             alternate;
}
.ball__wrapper {
  animation: horizontal 2s infinite ease-in-out
             alternate;
}
@keyframes horizontal {
  0% {
    transform: translate(-25vw, 0);
  }
  100% {
    transform: translate(25vw, 0);
  }
}
@keyframes vertical {
  50% {
    transform: translate(0, -20vh);
  }
}
```

Now check the demo. The ball travels on a curved path.



Vertical and Horizontal animations running in parallel.

Try changing the speed curve of those animations. How does it affect the ball's flight? **#MoveThingsWithCSS**

There is another technique for CSS animation along a curved path. It does require some familiarity with SVG path notation. Using the `offset-path` property, we can animate an element along a specified path. The caveat is that making these styles of animation responsive can be tricky.

CSS VARIABLES

CSS variables are awesome. But we can’t animate the values yet. Check this demo and you'll see how the value changes as if it has stepped timing. Can we use CSS variables with our animation? Yes. We can use their values to create dynamic animations.

Three medals that we want to animate.

Consider an example where we have three medals. We want to animate them all growing to different scales. Do we need three different animations for this? No. With CSS variables we can make the animation dynamic.

In this demo, each medal has a scoped CSS variable defining the scale for when it grows.

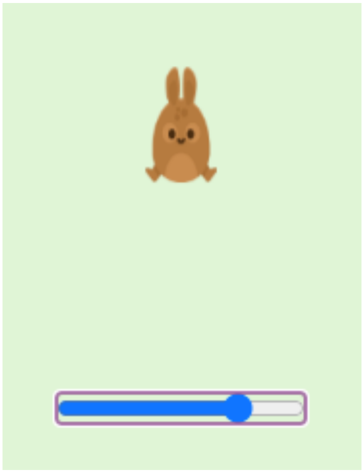
```
.medal {
  animation: scale 0.5s forwards;
}
@keyframes scale {
  to {
    transform: scale(var(--scale));
  }
}
.medal--bronze {
  --scale: 1.25;
}
.medal--silver {
  --scale: 1.5;
}
.medal--gold {
  --scale: 1.75;
}
```

Try experimenting with different scales. Can we have different starting scales in the animation too? **#MoveThingsWithCSS**

Before, you couldn't update an animation at run time by changing CSS variable values. The trick in this scenario would be to flip the animation on its head and update the initial state of an element.

```css
.bunny {
  transform:
        translate(0, calc(var(—height) * -1px));
}
@keyframes jump {
  50% {
    transform: translate(0, 0);
  }
}
```

Consider this demo where moving the slider changes the animation behavior. In this demo, the bunny jumps higher or lower.



The height is defined by a CSS variable.

Animations will respond to changing CSS variables at runtime now. We could update our bunny demo so that the slider updates the scale at the same time. We can change the animation code so that the `transform` responds to variable change.

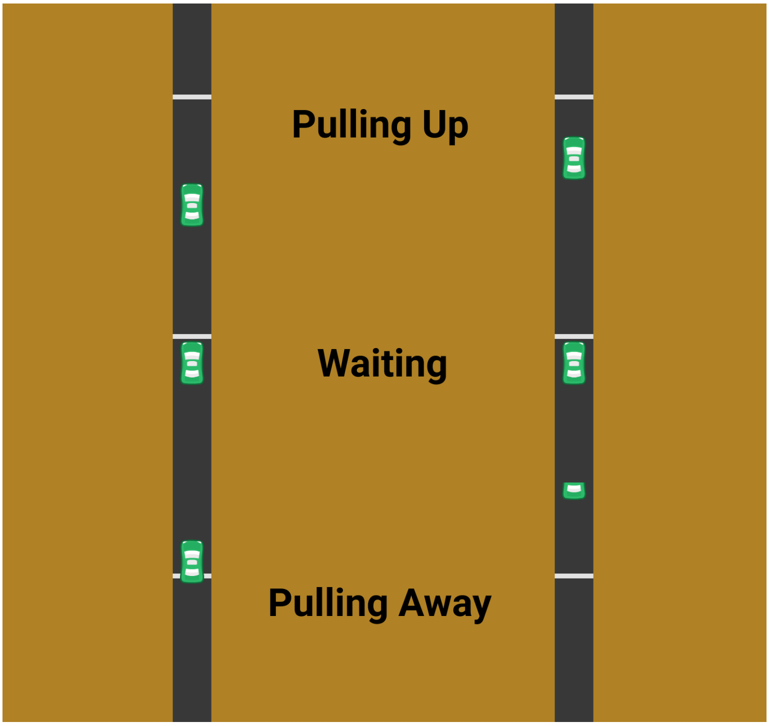
```
@keyframes jump {
  50% {
    background-image: url(“./bunny-jump.png");
    transform:
      translate(0, calc(var(--height) * -1px))
scale(calc(1 + ((var(--height, 1) / 100) * 2)));
  }
}
```

Now check the demo and move the slider. A greater value gives a higher jump with a larger scale. This opens up various opportunities for our animations.

Can you make an interesting animation that changes based on user interaction? **#MoveThingsWithCSS**

COMPOSITION

The composition of animations can become complex. Especially if we want to create timeline style animations. Imagine an animation timeline where a car pulls up to a crossing, waits, and then pulls away.

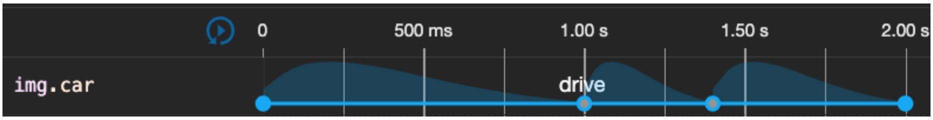


The timeline for our car.

We could create one set of keyframes and calculate a delay that we embed in the keyframes.

```
@keyframes drive {
  0% {
    transform: translate(0, 50vh);
  }
  50%, 70% {
    transform: translate(0, 0);
  }
  100% {
    transform: translate(0, -100vh);
  }
}
```

Check the demo. The car pauses from `50%` to `70%`. The `animation-duration` is set to `2s`. The car pauses from `1s` to `1.4s`. If the requirements change and we want a `1s` gap, we can extend the `animation-duration`. But It's a change that requires extra calculation.



The timeline for the car’s drive.

Or, we could create two animations and use variables to configure the timeline.

```
:root {
  --pull-up: 1;
  --wait: 1;
  --pull-off: 0.6;
}
.car {
  animation: pull-up calc(var(--pull-up) * 1s),
    pull-off calc(var(--pull-off) * 1s)
     calc((var(--pull-up) + var(--wait)) * 1s);
}
@keyframes pull-up {
  from {
    transform: translate(0, 50vh);
  }
}
@keyframes pull-off {
  To {
    transform: translate(0, -100vh);
  }
}
```

Check out the demo. Now variables dictate the car's journey.

Try changing the composition of the animation with the variables. Could you add another animation?
#MoveThingsWithCSS

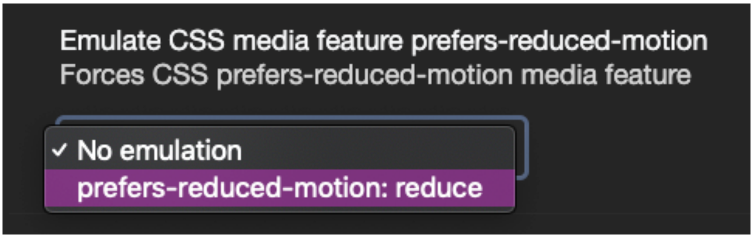
PREFERS-REDUCED-MOTION

What if our users would prefer reduced motion or don't want to see motion at all? This is very important for accessibility. We should consider users with vestibular motion disorders.

CSS provides a media feature that allows us to detect if a user has requested reduced motion.

```css
@media (prefers-reduced-motion) {
  .bunny {
    animation-name: none;
  }
}
```

This means we can either turn off animation completely or change out our animation. If we change the animation, try to create toned down versions that avoid large scaling or panning.



Emulate CSS media feature prefers-reduced-motion
Forces CSS prefers-reduced-motion media feature

The emulator option in Chrome Devtools.

Chrome's Devtools have the ability to emulate `prefers-reduced-motion`. Check out this demo and emulate it. With `prefers-reduced-motion` our bunny stops jumping. You'll find the emulate option in the "Rendering" tab which can be opened via "More tools".

JAVASCRIPT HOOKS

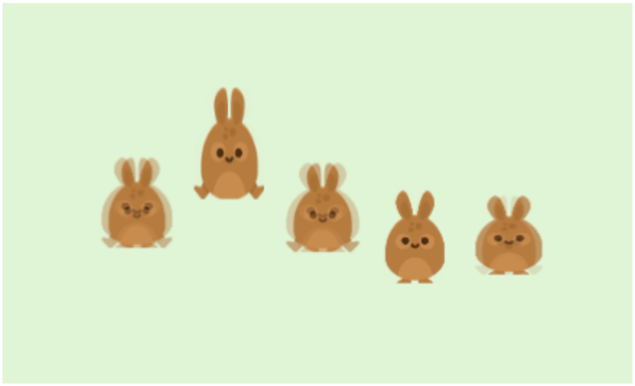
What if we need to our animation and transition lifecycles from our JavaScript? No worries, there are events we can hook into for both animations and transitions.

For transitions we have:

- `transitionstart` - triggered at the start of a transition.
- `transitioncancel` - triggered if a transition is cancelled. This could be a result of DOM or style changes.
- `transitionend` - triggered at the end of a transition.

For animations we have:

- `animationstart` - triggered at the start of an animation.
- `animationcancel` - triggered if an animation is cancelled. This could be a result of DOM or style changes.
- `animationend` - triggered when an animation ends.
- `animationiteration` - triggered after each animation iteration.


Those bunnies love jumping!

These hooks are pretty powerful and give us many opportunities. Consider our bunny demo from earlier. We could keep a track of how many times the bunny jumps.

```
const BUNNY = document.querySelector(‘.bunny')
const LABEL = document.querySelector('label')

let span = 0

const UPDATE = () => {
  span++
  LABEL.innerText = `Jumped ${span} times!`
}

BUNNY.addEventListener('animationiteration',
UPDATE)
```

Check out the demo. We now keep track of the number of jumps.

The hooks open up a bunch of opportunities. What if our bunny jumped, but after each jump took a little break? And we wanted to make that break a random length each time?

```
const UPDATE = () => {
  // Remove the animation
  BUNNY.style.animationName = 'none'

  // Create new break
  const INTERVAL = Math.floor(Math.random() *
5000)

  // After the break, set the animation name
  setTimeout(() => {
    BUNNY.style.animationName = 'jump'
  }, INTERVAL)

  // Update the label
  LABEL.innerText = `Jumping in ${(INTERVAL/
1000).toFixed(1)}s!`
}

BUNNY.addEventListener('animationend', UPDATE)
```

Check out the demo. Now the bunny jumps at random intervals.

JAVASCRIPT SOLUTIONS

If you've gone through this book, you should now know enough to get things moving with CSS. It might seem like an odd topic to end on but is CSS the right tool for you to move things? Yes and no. Hear me out. CSS animations and transitions are great. You can achieve so many wonderful things. But if your animations start to become complex, they become harder to maintain. This animation challenge series from Carl Schoof proved that. I attempted to complete the challenges with CSS variables.

What can we do when things get complex? We could make use of the JavaScript hooks to manage things better. But you don't have to start reinventing the wheel. Support for the Web Animations API is growing and there are polyfills. There are also great third party solutions for complete control over your animations. Here are some options:

- GreenSock (GSAP)
- Framer Motion
- Velocity.js

If there are JavaScript solutions, why learn CSS animation? The concepts and foundations translate. Reaching JavaScript first for animations can be overkill. It can sometimes add complexity or extra overhead. You might not be able to rely on JavaScript too. Your users may not always have JavaScript enabled. It's all about finding a balance and being able to judge when to reach for JavaScript, and when to stick with CSS. For example, CSS is great for loading spinners and certain micro-interactions. But, more complex timelines can be better handled in JavaScript.

CONCLUSION

We've covered a lot in a short space of time. By reading this book, you now have the skills to go and start making things move with CSS. Being able to animate your work is a valuable skill for making it stand out. We've built up a good foundation for how to get things moving with CSS animations and transitions. But we've only scratched the surface for the possibilities. Now it's time to go off and practice, practice, practice! That's the best way to hone your animation skills. I hope you've enjoyed reading this book and the material has inspired you. Inspired you to go and make wonderful experiences for others on the web! I can't wait to see them.

RESOURCES

I've linked out to and mentioned various resources along the way. Please do check them out, especially the demos. Here's a list of resources shared:

- "Move Things With CSS" Codepen Collection from Jhey Tompkins: https://codepen.io/collection/3c9c46ea96d05c765e3133109e8db922?grid_type=grid&sort_order=asc&sort_by=id
- "Twelve basic principles of animation", Wikipedia: https://en.wikipedia.org/wiki/Twelve_basic_principles_of_animation
- "Disney's motion principles in designing interface animations", Ruthi: https://medium.com/@ruthiran_b/disneys-motion-principles-in-designing-interface-animations-9ac7707a2b43
- "The ultimate guide to proper use of animation in ux", Taras Skytskyi: https://uxdesign.cc/the-ultimate-guide-to-proper-use-of-animation-in-ux-10bd98614fa9
- "Animatable CSS properties", MDN: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_animated_properties
- "Animatable" from Lea Verou: http://projects.verou.me/animatable/
- "High performance animations", Paul Lewis and Paul Irish: https://www.html5rocks.com/en/tutorials/speed/high-performance-animations/

- “Inspect animations”, Kayce Basques: https://developers.google.com/web/tools/chrome-devtools/inspect-styles/animations
- “CSS Triggers”: https://csstriggers.com/
- “Easings.net”: https://easings.net/
- “The basics of easing”, Paul Lewis: https://developers.google.com/web/fundamentals/design-and-ux/animations/the-basics-of-easing
- “Pseudo classes”, MDN: https://developer.mozilla.org/en-US/docs/Web/CSS/Pseudo-classes
- “animation property”, MDN: https://developer.mozilla.org/en-US/docs/Web/CSS/animation
- “Create a responsive CSS motion-path? Sure we can”, Jhey Tompkins: https://css-tricks.com/create-a-responsive-css-motion-path-sure-we-can/
- “Using the Web Animations API”, MDN: https://developer.mozilla.org/en-US/docs/Web/API/Web_Animations_API/Using_the_Web_Animations_API
- “No frills web animation sequencing challenge”, Carl Schoof: https://www.snorkl.tv/challenge/
- "Vestibular symptoms”, vestibular.org: https://vestibular.org/article/what-is-vestibular/symptoms/
- “Setting multiple animation property values”, MDN: https://developer.mozilla.org/en-US/docs/Web/CSS/CSS_Animations/Using_CSS_animations#Setting_multiple_animation_property_values
- “Easing function”, MDN: https://developer.mozilla.org/en-US/docs/Web/CSS/easing-function

ACKNOWLEDGEMENTS

I'd like to thank the group of people that joined me on this “Info Product Challenge”. It’s pushed me to create this content for you.

Thanks to the community for supporting and sharing my work. It means a lot and drives me to create more content like this.

And thank you! Thank you for picking up a copy of "Move Things with CSS".

STAY AWESOME!

ABOUT THE AUTHOR

Jhey Tompkins makes awesome things for awesome people! He's a developer with a wealth of experience known for sharing his creative and visual work online. He thrives on bringing ideas to life!

He writes about various topics and has written for various publications. Publications such as **CSS Tricks** and **Smashing Magazine**. He's also an **Egghead.io** instructor.

FIN.