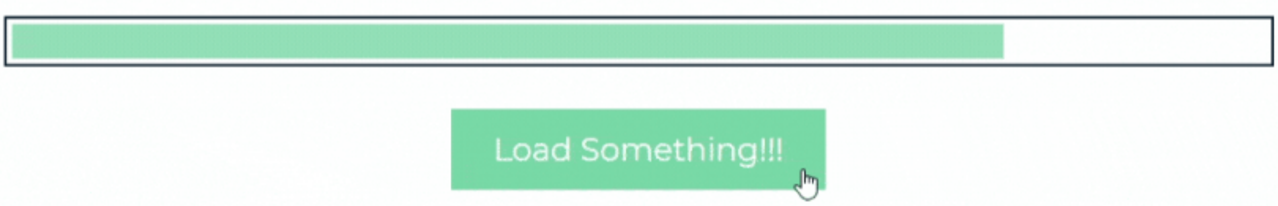
%%%
Contents 8i Create Modern CSS Animations **Panteli**
%%%

%%%
8i Part #1 - Get started with CSS Animations
%%%
0. About this course
- **notes on the video**
    - -> animations <- commonly used
    - -> for providing feedback / interactivity to users
    - -> tools / how to use them
    - -> transitions and gframes
    - -> optimising them / making them more efficient
- **notes on text below video**
    - about animations
        - -> manipulating the attention of the users
        - -> "the 12 principles of animation"
        - -> it's for UX
            - e.g clicking on a load button, and then a loading bar coming onto the screen



    - web animations are done with css
        - which means they can be done with Sass, because that compiles into css

        - -> Transform() <- this is a function
        - -> Pseudo-elements
            - -> a pseudos-selector is something for example, when you're defining a mixin (function)
            - -> the Python equivalent would be the argument, a variable whose value is only set inside the definition of the function
            - -> a pseudoclass e.g for something's hover state <- pseudo, something which applies some of the time but not all of the time
                - "pseudo... false apostle or teacher"
        - -> @keyframes <- to build a multi-stage animation
        - -> opacity, iteration and direction properties
            - -> an animation is a load of little pictures all put together -> little dx's
            - -> you're iterating through them -> depending on whatever logic condition happens
                - -> the zeotrope -> it shines a light and then iterates through different images, to make an animation
                - -> iteration and direction properties
            - -> opacity -> is weather it's transparent or not, which can change
            - -> direction properties -> which direction it's moving or e.g rotating in
        - -> user interactivity
        - -> CSS transitions, complex CSS transitions

1. Get the most out of this course

- **notes on video**
    - -> animating with CSS
        - -> the entire course is on animations with CSS
    - -> the structure of the course
        - three parts -> those are split into chapters
        - -> those chapters each have a video which explains the structure of its content
        - -> then there are quizzes and review sections
        - -> exercises
- **notes on text below video**
    - to make the most out of the course
        - the chapter video
            - -> a view of the contents
        - the text below the videos
            - -> how to implement the concepts
            - -> you can practice with this
        - quizzes
            - -> these are at the end of each of the third of the course
            - -> there are also codepen exercises

2. Get acquainted with animation for the web

- **notes on video**
    - -> css animations
    - -> making webpages have animations on them
        - for the attention of the users
        - the identity of the webpage <- to e.g be in line with the brand for the webpage
        - -> to make transitions e.g between pages smoother
- **notes on text below video**
    - the concept of an animation
        - -> CSS animations <- adding them to webpages
        - -> there is programming behind animations <- integrating them into webpages

        - -> they make objects move
    - body language
        - -> the idea that the object on the webpage is a part of it
        - -> and then by moving that object with an animation -> it's communicating with the "body language" of the webpage
        - -> then he's taking about the importance of body language
        - -> in other words, using text to communicate on the webpage, vs animations
            - Mehrabian, Silent Messages book <- communication is 7% verbal, 38% tone, and 55% body language
    - micky mouse
        - cel animation
            - it's occupying a square
            - -> these were hand drawn, and were used to make animation use more mainstream
        - 3D animation
        - claymation
            - -> talking raisins
    - possibilities
        - -> this course is adding animations to webpages, using CSS
        - -> using CSS to -> over time, change the appearance of webpages
        - example he's giving of this are
            - -> hovering over a button and the background colour of the button transitioning in -> rather than just showing
            - -> when the user enters the wrong password, shaking the password box
            - -> when elements come onto the webpage -> making them transition in from the left
                - like a power point transition but for webpages -> done with css
                    - you're power pointing in, for example the background colour of a button or an entire section of elements
    - more about why they work
        - -> psychology <- people notice motion
            - part of communication
            - attention economy
            - "fluid" user experience

3. Use CSS transitions for simple animations

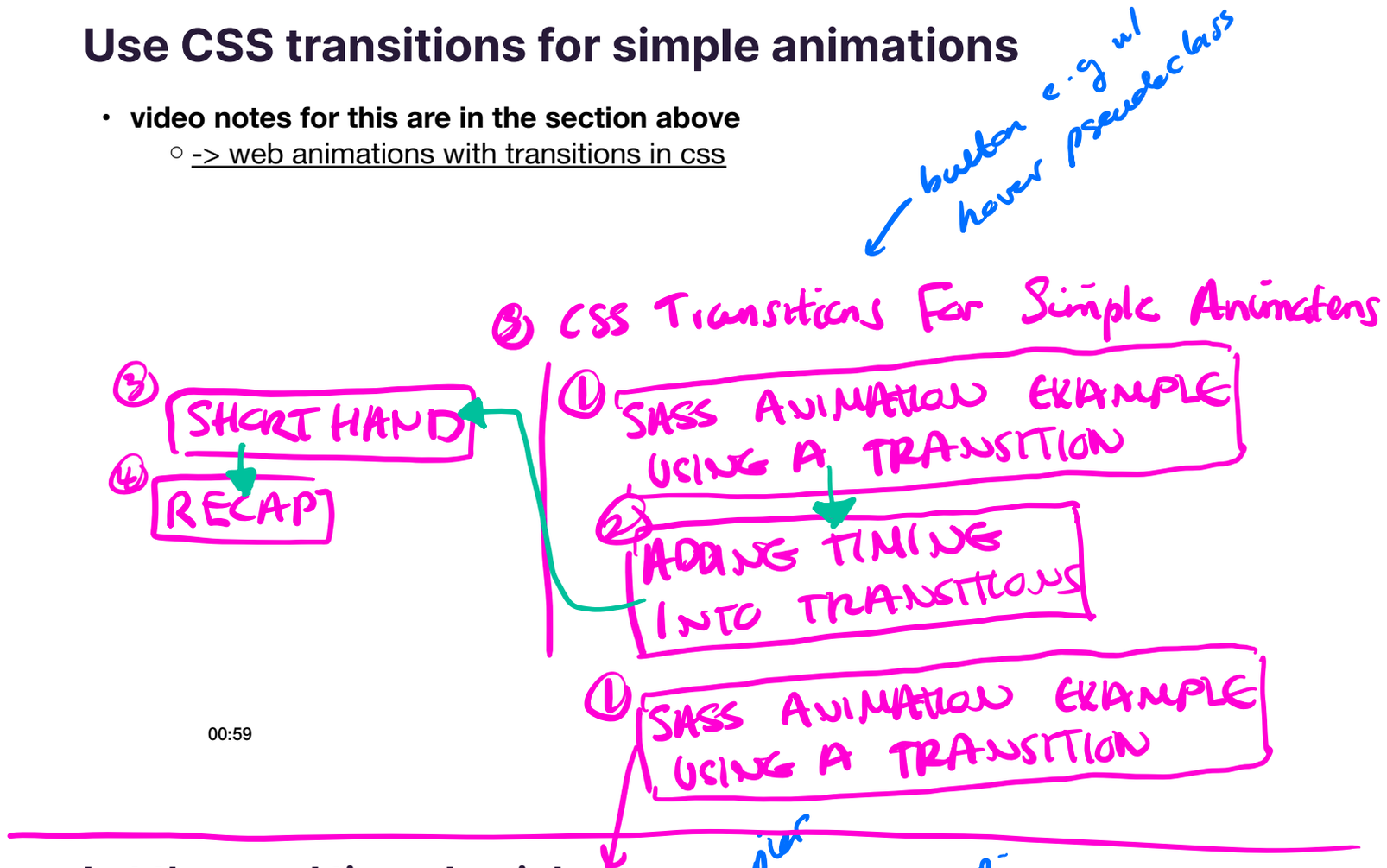
- **video notes**
    - -> creating animations
    - -> CSS transitions <- animating CSS properties
        - e.g to change the size of the button / make something move across the screen

Create Modern CSS Animations

3. Use CSS transitions for simple animations

Use CSS transitions for simple animations

- **video notes for this are in the section above**
  - -> web animations with transitions in css



Let the good times begin!

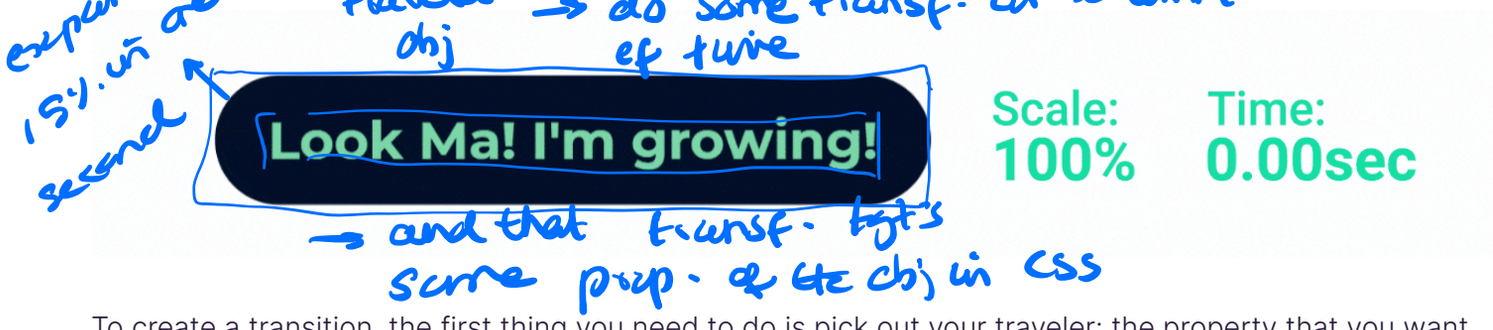
Alright. Enough talking, let's start animating!

Animations in CSS — simpler — complex

To create animations with CSS, you use either **transitions** or **keyframes.** While keyframes require a bit more to get set up and running, transitions only require a few bits of info to create. They are perfect for quick, simple animations, such as changing the appearance of a button when the user hovers over it.

Let's start with transitions which allow you to change the value of a property over a set amount of time. When you think of the word "journey," what comes to mind? Probably a traveler who follows a route from one point to another, taking a few minutes, hours, or days to complete the trip. These are the most basic components of an animation.

The traveler represents the CSS property you'd like to change. The starting point is the initial value of the animated property, the endpoint is its final value, and the time to complete the journey is its duration. Let's say you want to increase a button's scale by 15% over the course of a second:



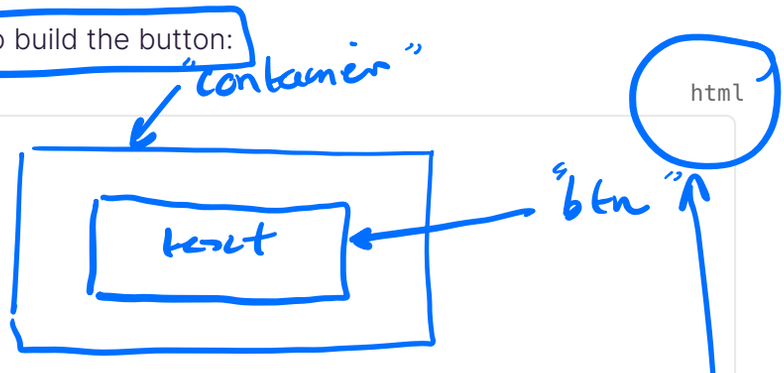
To create a transition, the first thing you need to do is pick out your traveler: the property that you want to animate. Since you want your button to grow when a user hovers over it, the `transform` property and its `scale()` function is a good choice to achieve the effect:

```scss
transform: scale(1.0);
```

> Don't worry if you're not familiar with the `transform` CSS property, and its plethora of functions! We'll go over it in more detail in just a bit. For now, just know that you can use the `scale()` function to change an element's size, based on the value you give it, where 0=0% and 1=100%.

Let's take a quick look at the HTML we've used to build the button:

```html
<body>
    <div class="container">
        <div class="btn">
            Look Ma! I'm growing!
        </div>
    </div>
</body>
```



It's made of a div with the class `.btn` applied to it, filled with text for the button's contents. Here's the Sass used in the `.btn` class to define its color, padding, and sizing:

```scss
$cd-btn: #011c37;
$cd-txt: #15DEA5;

.btn {
    background: $cd-btn;
    color: $cd-txt;
```

```
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
}
```

Now that we've chosen the `transform` property as the traveler, we need to send it on a journey, where we start with a scale of `1.0` (100%) and grow it to a scale of `1.15` (115%) when the button is hovered over.

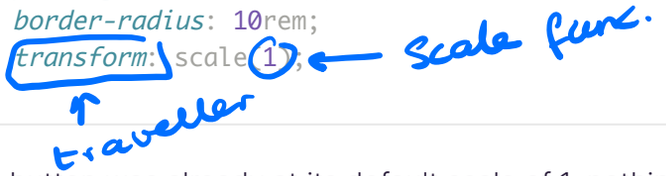
Getting from point A to point B: transitions

So, let's add `transform` and it's `scale()` function to the `.btn` class:

scss

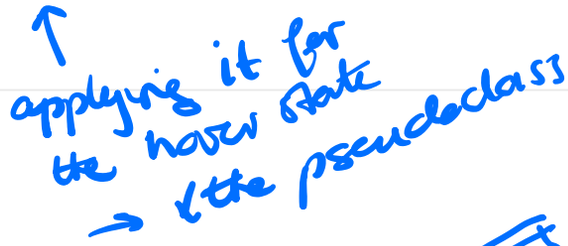
```scss
$cd-btn: #011c37;
$cd-txt: #15DEA5;

.btn {
    background: $cd-btn;
    color: $cd-txt;
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
    transform: scale(1);
}
```



Since the button was already at its default scale of 1, nothing much has changed:

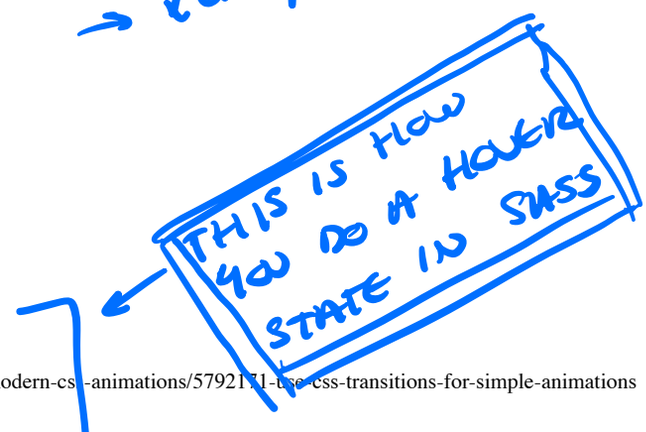
That's because we haven't given our trip it's destination yet. Since we want the button to increase in size when hovered over, let's add a `:hover` pseudo-selector, with the transform's `scale()` set to `1.15`:



scss

```scss
$cd-btn: #011c37;
$cd-txt: #15DEA5;

.btn {
    background: $cd-btn;
    color: $cd-txt;
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
    transform: scale(1);
    &:hover {
```



```
        transform: scale(1.15);
    }
}
```

Transitions need to be triggered by a change of state, which is typically done through the use of pseudo-classes, such as `:hover`. Now, when we demo the button, we see our starting and ending points, but there's no animation happening; the button just jumps between the two scales:

We need to tell the browser that we want the change in scale between the inactive and `:hover` states to animate as a transition. We can do by adding the `transition-property` property to our `.btn` selector, naming `transform` as the property that we want to transition:

```scss
$cd-btn: #011c37;
$cd-txt: #15DEA5;

.btn {
    background: $cd-btn;
    color: $cd-txt;
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
    transform: scale(1);
    transition-property: transform;
    &:hover {
        transform: scale(1.15);
    }
}
```

Okay! Now we've added a transition to the button. Let's check it out!

Gah! We've added the `transition-property`, but it's still not animating! What are we missing? Remember, animation is a change over *time*. We haven't told the browser how long to make the transition, so it assumes a duration of 0 seconds, which isn't much of an animation.

If you encounter a problem where your transition doesn't seem to be triggering, take a look at where you've placed your transition properties. They should be within the selector that contains the element's starting place, but it's common to accidentally place them within the selector for the end value instead, such as the `:hover` selector.

Are we there yet? — timing

Let's give our transition a length by using the `transition-duration` property and assigning it a value for the amount of time we'd like to take to complete the transition.

scss

```scss
transition-duration: 1s;
```

Within CSS, you can write time in two different ways; in seconds, as we've done above, or with the number of seconds, followed by the letter "s" to denote seconds. But you can also write it in milliseconds, where 1000 milliseconds is equal to 1 second, followed by the letters "ms" to denote milliseconds:

scss

```scss
transition-duration: 1000ms;
```

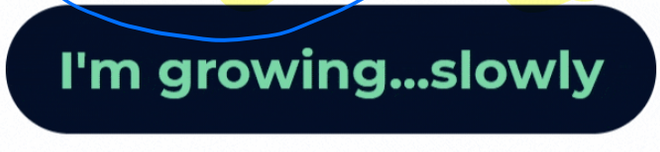
Both methods will produce the same results, so it's a matter of personal preference as to which format you use. Let's go with milliseconds and add a duration of 1000ms to our .btn selector:

scss

```scss
$cd-btn: #011c37;
$cd-txt: #15DEA5;

.btn {
    background: $cd-btn;
    color: $cd-txt;
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
    transform: scale(1);
    transition-property: transform;
    transition-duration: 1000ms;
    &:hover {
        transform: scale(1.15);
    }
}
```

And now we have motion!



Perfect! At least it's growing. But it might take longer to run its course. While there are always exceptions, you generally want interactive animations to be short enough that they feel seamless to the user. Right now, you have to hover and then wait to finish the transition. So, let's reduce the `transition-duration` to something like 400ms, and see how that feels:

scss

```scss
$cd-btn: #011c37;
$cd-txt: #15DEA5;

.btn {
    background: $cd-btn;
    color: $cd-txt;
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
    transform: scale(1);
    transition-property: transform;
    transition-duration: 400ms;
    &:hover {
        transform: scale(1.15);
    }
}
```

Let's test the button again:

Perfect! The button is growing 15% over the course of 400 milliseconds. You've just built your first CSS animation!

Virtual High Five

> There's no magical number for animation duration. You'll need to experiment to find the right length of time. Through enough practice, you will begin to get a feel for how long things should take, making finding the proper duration less time-consuming.

I'll be brief: shorthand properties

Now, remember when I said this?

> Which we can do by adding the `transition-property` property to our `.btn` selector

I used a keyword there: ***can.*** As in there are other ways of declaring a transition. Rather than typing out each property individually:

scss

```
transition-property: transform;
transition-duration: 400ms;
```

You can merge them all into a single, more concise property called `transition`, that contains all of the transition properties as a single list, starting with the name of the property, followed by a space and the duration:

scss

```
transition: transform 400ms;
```

Both formats produce the same result, but writing out transitions via the `transition` property and its shorthand is much more common, and what we'll be using from here on out. It's more concise and keeps the various properties grouped in a single location.

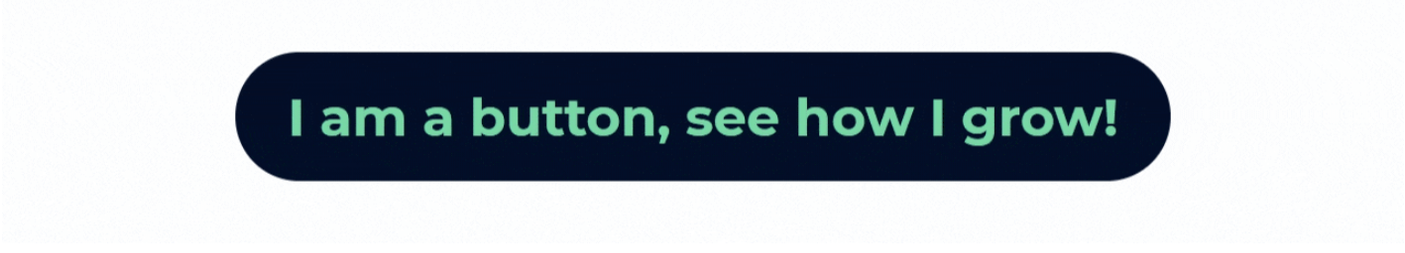
So, let's refactor the `.btn` code using the transition shorthand rather than individual properties:

scss

```
$cd-btn: #011c37;
$cd-txt: #15DEA5;

.btn {
    background: $cd-btn;
    color: $cd-txt;
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
    transform: scale(1);
    transition: transform 400ms;
    &:hover {
        transform: scale(1.15);
    }
}
```

Now we've tightened up the code, and everything still functions as it did before:



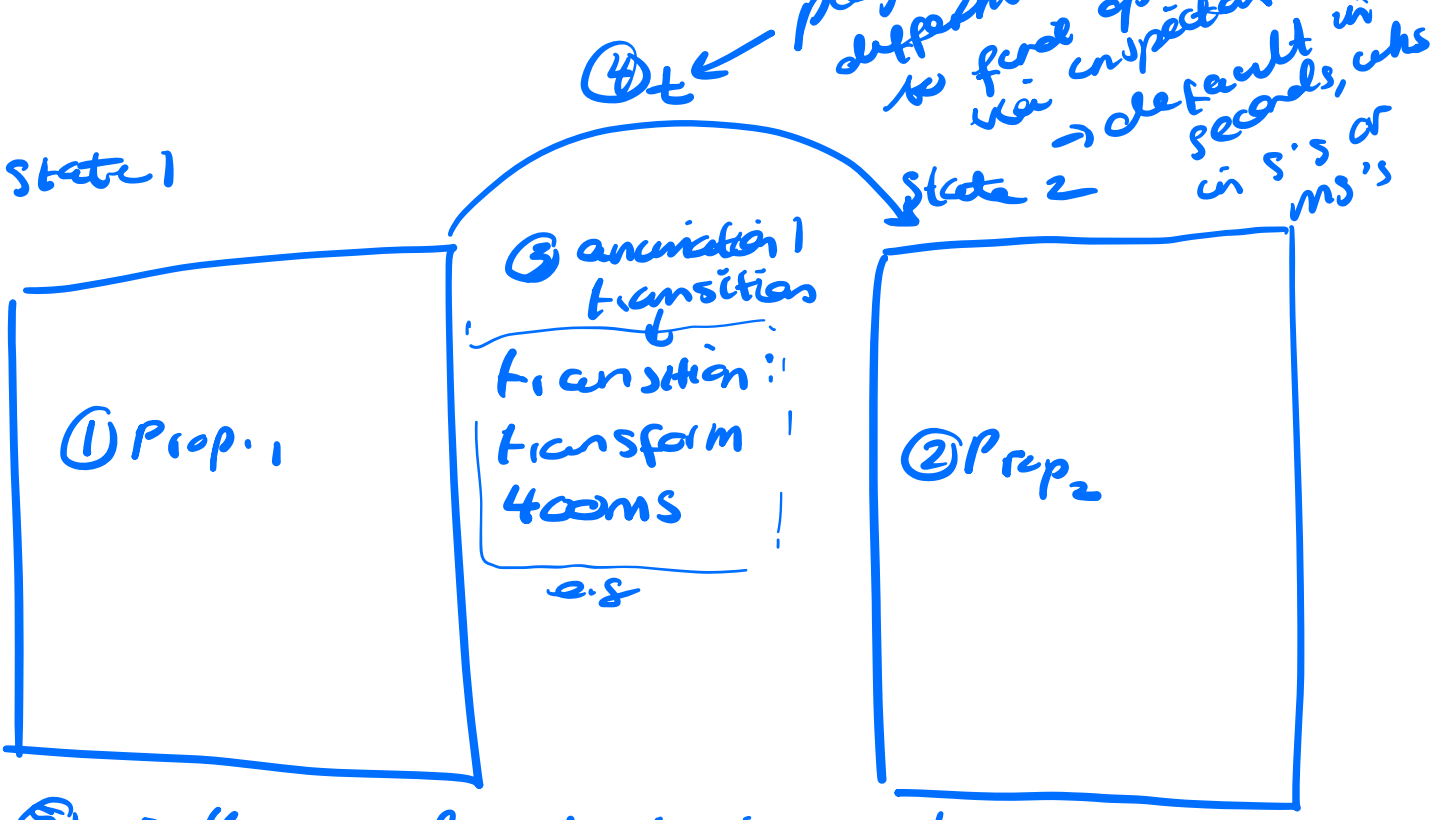
We've just used the `:hover` pseudo-selector to trigger button-growing transition, but there's a bunch of other pseudo-classes you can use with transitions. We'll dive into those next!

Let's recap!

- The 4 ingredients to a transition are:
  - A property to animate.
  - The before and after values of the property.

    - The duration of the transition.
    - A pseudo-selector to trigger the animation.
- Apply the before value to the element that you want to transition and the after value to the pseudo element that triggers the transition.
- Write the duration in either seconds: `.4s` or milliseconds: `400ms`.
- Write out the transition properties individually, ex: `transition-duration: 400ms`.
- Or combine them into a single transition property: `transition: transform 400ms`.

always proceed w/ this set values of or



⑤ → You need a triggering event - e.g - button is hovered over

→ This is done via a Sass pseudoselector (e.g : hover)

Create Modern CSS Animations

4. Use pseudo-selectors to trigger CSS transitions

Use pseudo-selectors to trigger CSS transitions

- -> hover pseudoclasses to trigger transitions
- -> you can trigger transitions when something is hovered over
- -> you can also use pseudoclasses to trigger animations / transitions on other objects on the webpage
- -> how to use hover states to animate transitions on the element which is being hovered over and over the other elements on the webpage (which aren't being hovered over, but which change when another element on the webpage is being hovered over)
  - -> and how they change is via transitions done in css / Sass (indented css)
  - -> that is done with transitions -> aka animations in css / Sass
  - -> there are also - think they were called key frames?
    - But at the moment we're still looking at transitions

CONTENTS FOR USING PSEUDO-SELECTORS TO TRIGGER CSS TRANSITIONS CHAPTER

① TRIGGERING TRANSITIONS

② EMAIL VALIDATION EXAMPLE

③ TRIGGERING SIBLINGS

④ RECAP

06:26

Lighting the fuse: triggering transitions

① TRIGGERING TRANSITIONS

One of the key ingredients to creating CSS transitions is incorporating a pseudo-class to allow the user to change an element's state and trigger a transition. We've just used a `:hover` pseudo-selector to trigger the growth of a button. And, while `:hover` is probably the most common pseudo-class that you'll use, there are others that can trigger transitions as well.

→ :hover isn't the only pseudoclass
→ when it's being hovered over, e.g. expand the

In fact, there are more than 50 pseudo-classes to choose from, but not all of them are well suited to transitions. To understand which will, or won't, work well with transitions, let's take a look at what a pseudo-class is and how it works.

A pseudo-class is a lot like an if-statement for the browser: If the condition of the pseudo-class is true, then apply its styling to the object. In the case of our growing button, if it's being hovered over, use the styling contained within the `:hover` pseudo-selector, such as transforming it by increasing its scale.

Hovering over the button changes its state from "not-hovered" to "hovered." It is that change that makes it perfect for triggering transitions. The best pseudo-class candidates are those whose states you can anticipate being changed by the user as they interact with the site. Just like you can anticipate a user hovering over a button, you could also anticipate them making a text-input `:active`, or entering text that is `:invalid` and style them as you see fit.

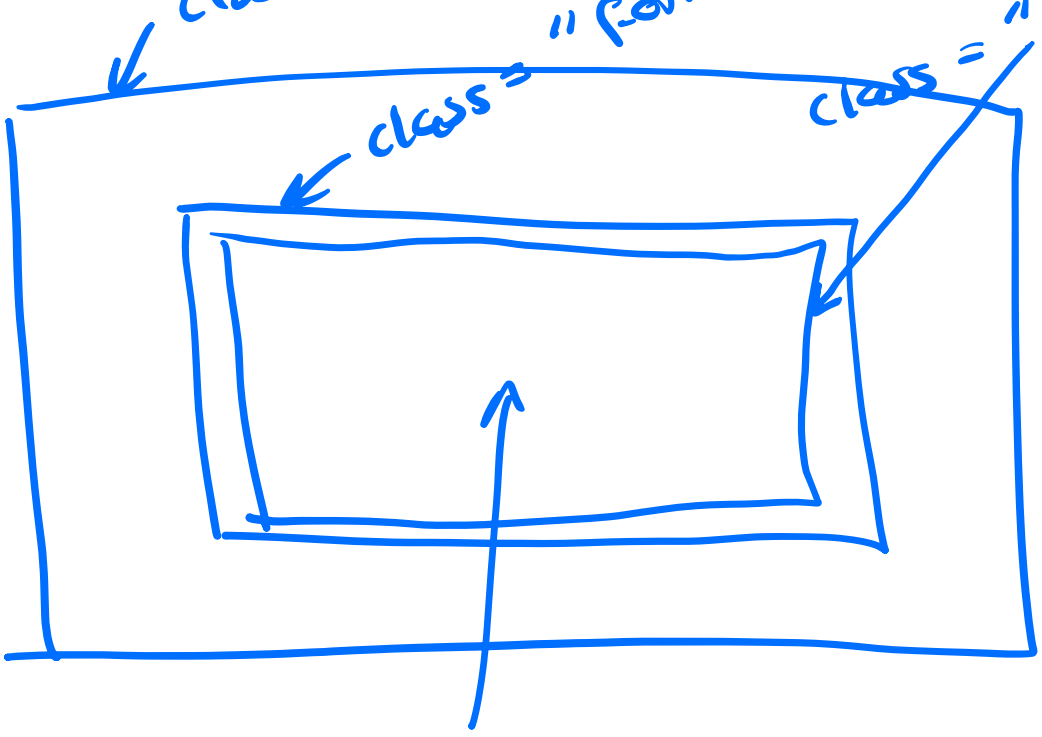
The following are pseudo-classes commonly used in conjunction with transitions. That isn't to say they are the only ones, but rather a list of likely candidates. For a complete list of pseudo-classes, check out the MDN documentation. Even those that don't work well with transitions can make styling elements a lot easier.

- :hover
- :active
- :focus
- :valid
- :invalid
- :not()
- :checked
- :enabled
- :disabled

Constructive criticism: validation

Let's create a form that gives the user feedback as they enter their email address. The HTML for our form could look something like this:

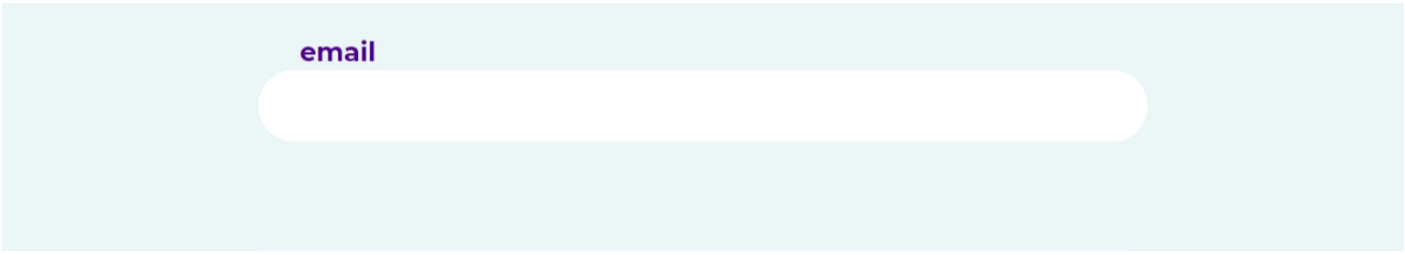
```html
<body>
    <div class="container">
        <div class="form">
            <div class="form__group">
                <label for="">email</label>
                <input type="email" name="" id="">
            </div>
        </div>
    </div>
</body>
```



It's asking for an email to be entered there

↑ This is what the html is creating

We have `.form` block element, containing a `.form__group` , which is comprised of an email input and it's label, which renders in the browser as:

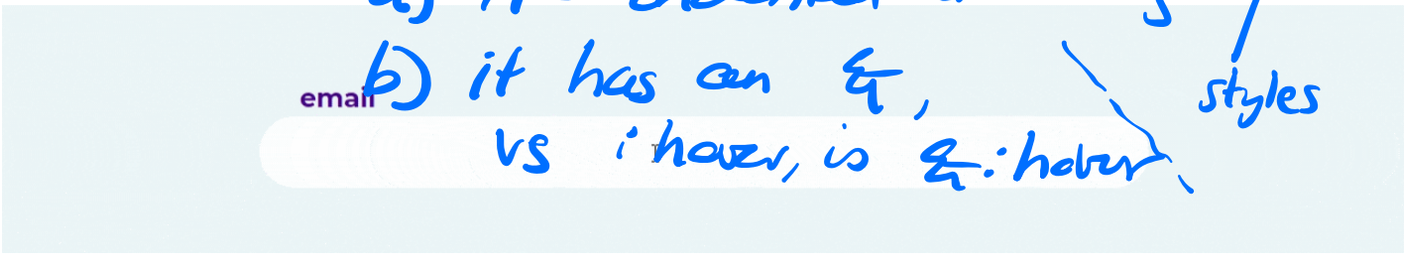


Now, let's use the `:active` pseudo selector to provide the user with a bit of feedback, letting them know that they have made the input field **active** by highlighting it with a border:

```scss
$cd-txt: #6300a0;

.form {
    &__group {
        & input {
            border: 2px solid $cd-box;
            border-radius: 100rem;
            color: $cd-txt;
            font-family: 'Montserrat', sans-serif;
            font-size: 2.5rem;
            outline: none;
            padding: .5rem 1.5rem;
            width: 100%;
            &:focus {
                border: 2px solid $cd-txt;
            }
        }
    }
}
```

Now, when we change the state of the input to `:focus` , it should add a purple border to the element:



Perfect! Now, let's use HTML5's built in ability to validate fields, and combine it with the `:invalid` pseudo-class, so that they receive feedback that they haven't entered a valid email address. Let's start by adding an `:invalid` pseudo-selector to the input selector and add properties to change its text to white on a red background:

```scss
$cd-txt: #6300a0;
$cd-txt--invalid: #fff;
```

```scss
$cd-danger: #b20a37;

.form {
    &__group {
        & input {
            border: 2px solid $cd-box;
            border-radius: 100rem;
            color: $cd-txt;
            font-family: 'Montserrat', sans-serif;
            font-size: 2.5rem;
            outline: none;
            padding: .5rem 1.5rem;
            width: 100%;
            &:focus {
                border: 2px solid $cd-txt;
            }
            &:invalid {
                background: $cd-danger;
                border: 2px solid $cd-danger;
                color: $cd-txt--invalid;
            }
        }
    }
}
```

Now let's take our form for a spin!

email

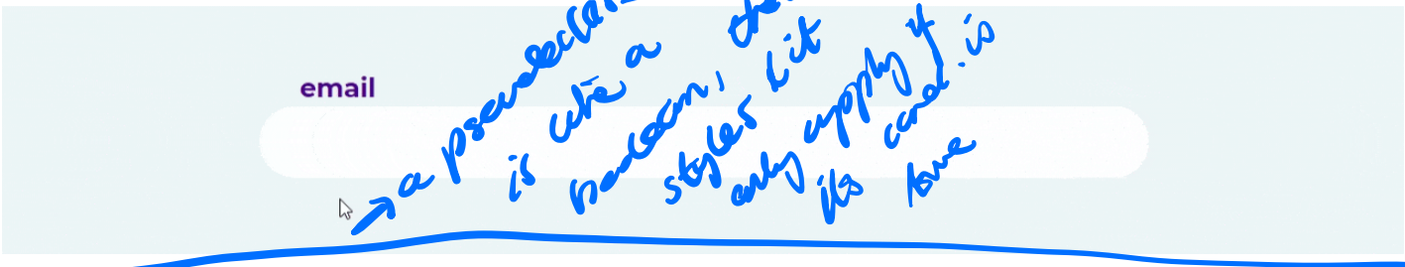
It's validating the email address as it's being entered, but flickering between the `:focus` and `:invalid` is a bit obnoxious. Let's make use of the `:not()` pseudo-selector, combined with the `:focus` pseudo-selector to allow the user to finish entering their email address before telling them whether it's valid or not:

```scss
$cd-txt: #6300a0;
$cd-txt--invalid: #fff;
$cd-danger: #b20a37;

.form {
    &__group {
        & input {
            border: 2px solid $cd-box;
            border-radius: 100rem;
            color: $cd-txt;
            font-family: 'Montserrat', sans-serif;
            font-size: 2.5rem;
            outline: none;
            padding: .5rem 1.5rem;
            width: 100%;
```

```
16          &:focus {
17              border: 2px solid $cd-txt;
18          }
19          &:not(:focus):invalid {
20              background: $cd-danger;
21              border: 2px solid $cd-danger;
22              color: $cd-txt--invalid;
23          }
24      }
25  }
26  }
```

The `:not()` pseudo-selector registers as true when it's containing selector registers as false. In this case, `:not()` will return true if the input is **not** in the `:focus` state. You can chain pseudo-classes together, just like with standard classes. In this case, we've created a pseudo-selector that will apply its rule set if the element isn't in the `:focus` state **and** the email address is `:invalid`. This way we don't see the red warning signs of an invalid email until they've finished typing their address:



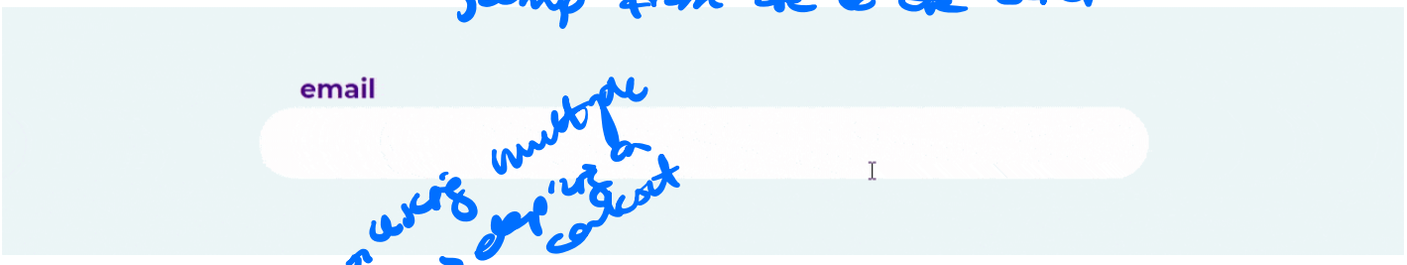
There. That's much nicer, don't you think? But we can make it nicer yet! This is an animation class, after all, so let's add animation to the changes in state by adding a transition for the `background-color` that takes half of a second to complete:

scss

```
1  $cd-txt: #6300a0;
2  $cd-txt--invalid: #fff;
3  $cd-danger: #b20a37;
4
5  .form {
6      &__group {
7          & input {
8              border: 2px solid $cd-box;
9              border-radius: 100rem;
10             color: $cd-txt;
11             font-family: 'Montserrat', sans-serif;
12             font-size: 2.5rem;
13             outline: none;
14             padding: .5rem 1.5rem;
15             width: 100%;
16             transition: background-color 500ms;
17             &:focus {
18                 border: 2px solid $cd-txt;
19             }
20             &:not(:focus):invalid {
21                 background-color: $cd-danger;
22                 border: 2px solid $cd-danger;
23                 color: $cd-txt--invalid;
```

```
24          }
25      }
26  }
27 }
```

Now we'll get a quick fade between when the user enters their email and when it displays as invalid:

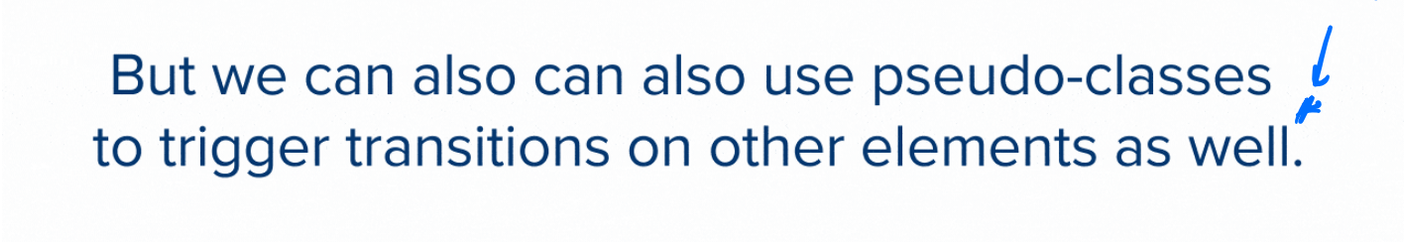


Perfect! 👌 By chaining together pseudo-classes into a more specific pseudo-selector, we've been able to improve the user's experience!

Picking on your little brother: triggering siblings

Now, as you've just seen, you can use pseudo-classes to trigger transitions on their assigned element, such as changing the size of a hovered button, or invalid text field. But you can also use pseudo-classes to trigger transitions on *other* elements as well.

Hey Jim, yeah... can we get an asterisk for that last sentence? Perfect... thanks!



That's better. So, why the asterisk? What's the disclaimer?

For a pseudo-selector to trigger a transition on *another* element, that element **must** be the next sibling in the HTML document. In other words, you must use the adjacent sibling combinator: `+` when setting up the transition.

Say whuuh?

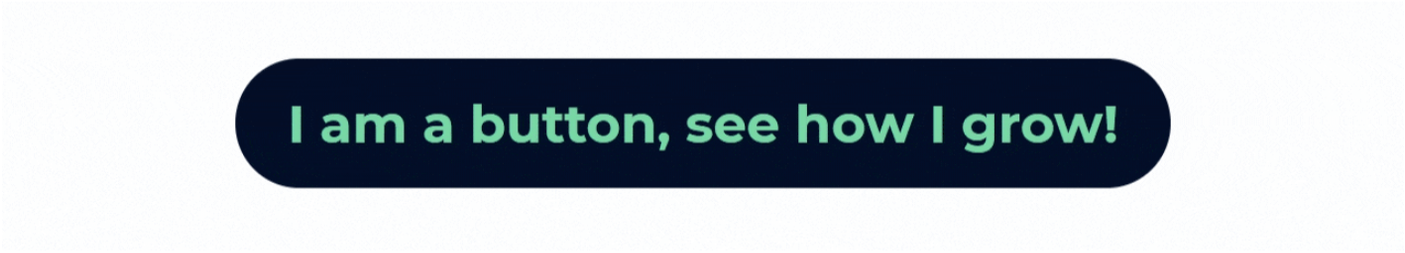
Use a `:hover` pseudo-selector and add styles to it like so:

scss

```
1 .btn {
2     background: $cd-primary;
3     font-size: 3rem;
4     cursor: pointer;
5     padding: 1.85rem 3rem;
6     border-radius: 10rem;
7     &:hover{
8         transform: scale(1.15);
9     }
```

```
10 }
```

In the event of a hover, those styles are applied to the element that `:hover` is attached to:



But let's say the button has a sibling with the class `.ball` assigned that you would like to affect instead.

there is a trigger (the element being hovered over), and then there is the thing being affected (moving from one state to another) — which is when the animation / transition happens (on it)

html

```
<body>
    <div class="container">
        <div class="btn">
            Grow!
        </div>
        <div class="ball"></div>
    </div>
</body>
```

You can use the adjacent sibling combinator to couple `.ball` with the `:hover` the pseudo-selector instead:

scss

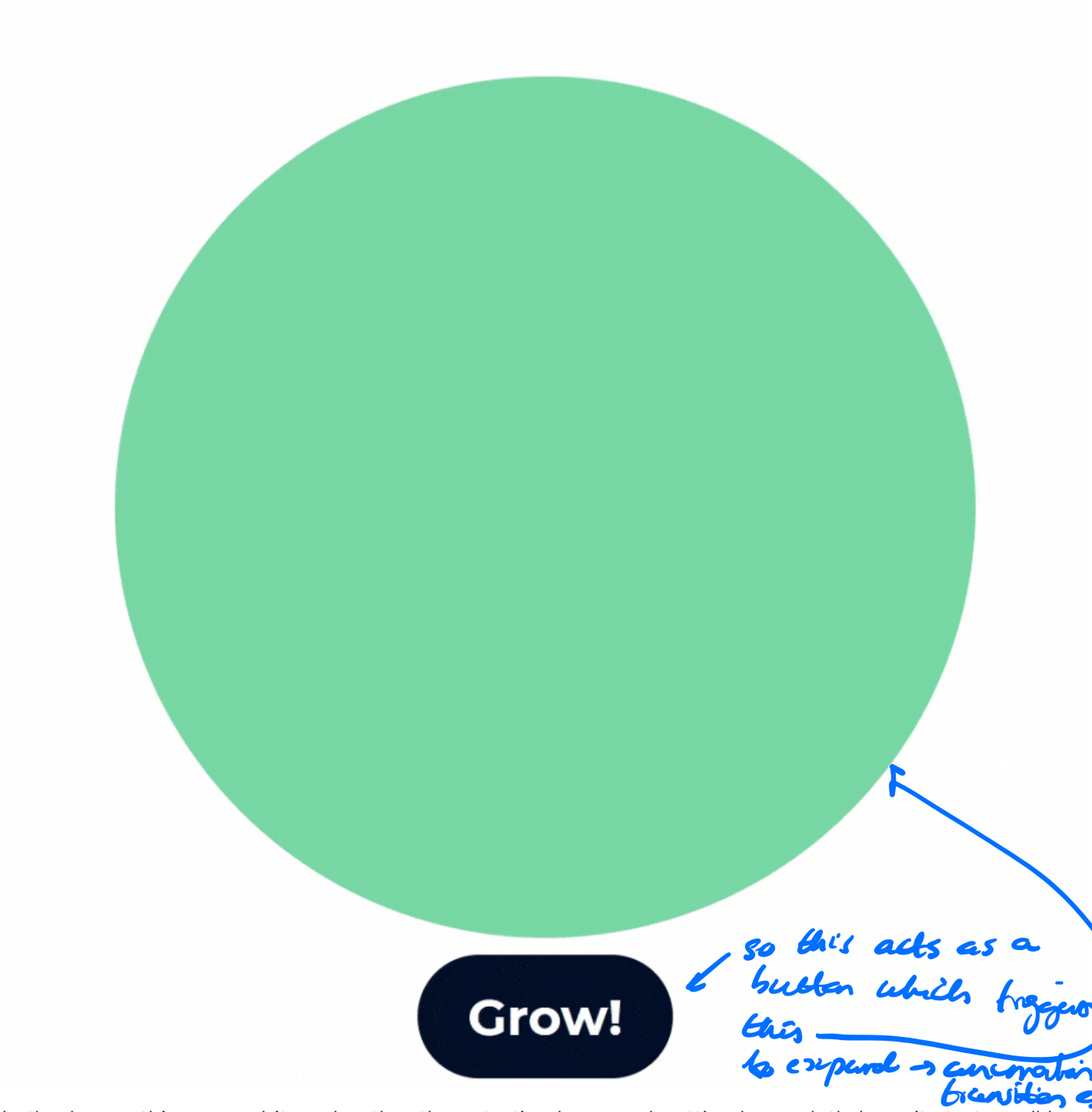
```
.btn {
    background: $cd-primary;
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
    &:hover + .ball{
        transform: scale(1.15);
    }
}

.ball {
    width: $ball-size;
    height: $ball-size;
    background: $cd-secondary;
    margin-bottom: 1rem;
    border-radius: $ball-size * 0.5;
}
```

You have code for the ball under code for the button
→ "when the button is being hovered over, apply this transition to the ball"
→ in this case the ball is a circle

that transition for the ball - the code for it is listed under the element being hovered over, not the one being affected

Now when you hover over the button, `.ball` will scale up instead:



Let's change things up a bit, and, rather than starting huge and getting huger, let's have it start small by setting the initial `scale()` to `.1`, and add a transition. But rather than taking a fraction of a second, let's have it take a full four seconds to complete the animation so it creates the perception of continuously growing:

scss

```scss
.btn {
    background: $cd-primary;
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
    &:hover + .ball{
        transform: scale(1.15);
    }
}

.ball {
```

```
    width: $ball-size;
    height: $ball-size;
    background: $cd-secondary;
    margin-bottom: 1rem;
    border-radius: $ball-size * 0.5;
    transform: scale(0.1);
    transition: transform 4000ms;
}
```

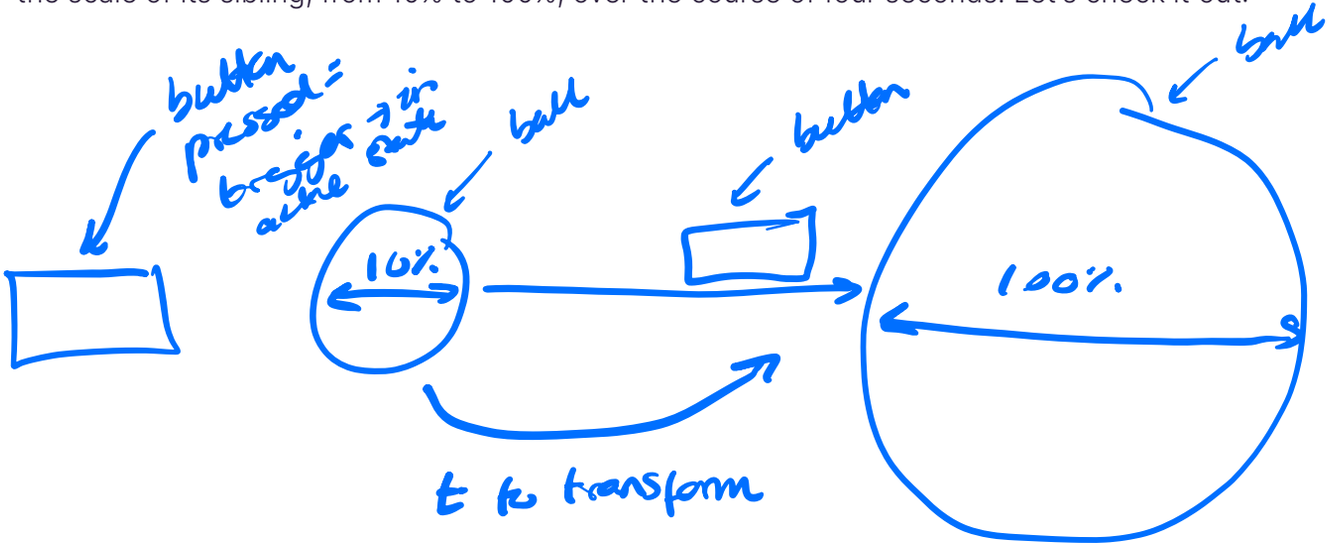
And, finally, let's change the `:hover` selector to `:active`, so the scale increases when the button is pressed, rather than simply hovered over. Then, change the scale of the `:active` state to be `1.0` rather than `1.15` and we should be all set!

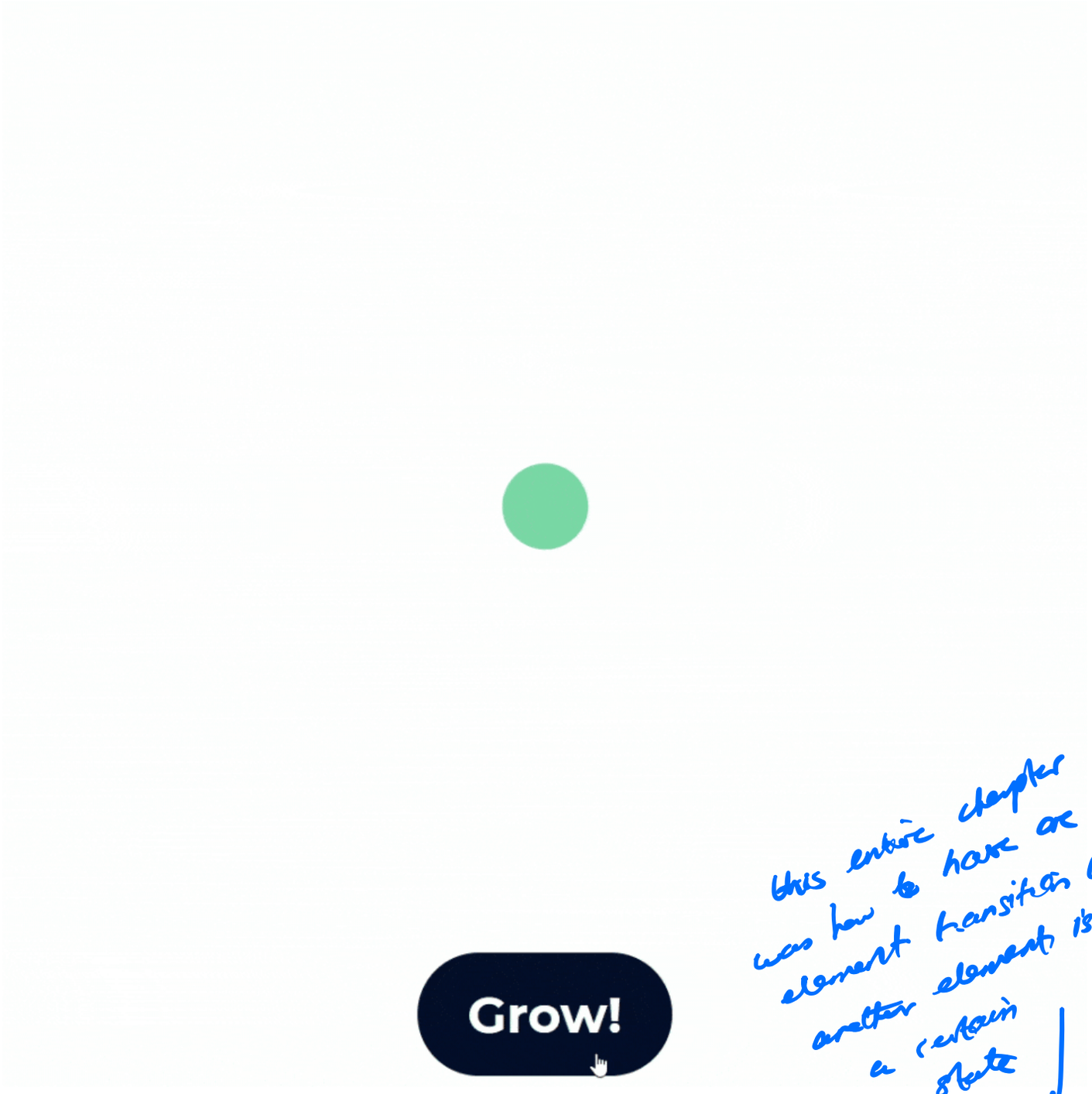
scss

```
.btn {
    background: $cd-primary;
    font-size: 3rem;
    cursor: pointer;
    padding: 1.85rem 3rem;
    border-radius: 10rem;
    &:active + .ball{
        transform: scale(1.0);
    }
}

.ball {
    width: $ball-size;
    height: $ball-size;
    background: $cd-secondary;
    margin-bottom: 1rem;
    border-radius: $ball-size * 0.5;
    transform: scale(0.1);
    transition: transform 4000ms;
}
```

When we view it in the browser now, we should have a button that, when pressed, triggers a transition to the scale of its sibling, from 10% to 100%, over the course of four seconds. Let's check it out:





Pseudo-classes are extremely helpful in styling elements and triggering transition based on their states, but can also be used to manipulate other elements on the page, allowing you to create more interactive experiences for your visitors.

We've covered the basics of transitions, so, coming up next, you'll learn methods for adding complexity and life to animations from some of the masters!

Let's recap!

- Pseudo-classes are used to trigger CSS transitions.
- The best pseudo-classes to use with transitions are those that the user will interact with as they browse the site.
- You can chain pseudo-classes together to make more specific pseudo-selectors.
- Pseudo-selectors can be used to manipulate its element's next sibling as well as itself.

↳ you can use multiple pseudoclasses for one el e.g hover / active state → and one el being in one state can be cond'l to run a transition on another el wd a pseudoclass which corrects them

Create Modern CSS Animations

5. Apply the 12 principles of animation to the web

CONTENTS → APPLY THE 12 PRINCIPLES OF ANIMATION TO THE WEB

① VIDEO NOTES
② MOVING ON FROM ONLY USING TRANSITIONS IN CSS FOR ANIMATIONS
③ THE NINE OLD MEN
④ RECAP

Apply the 12 principles of animation to the web

① VIDEO NOTES

- Animations in the previous chapters were done using transitions
  - Those transitions were put in with Sass (aka indented css done with an npm extension to make css look more like an OOP language rather than a scripting language)
  - -> this chapter is how to make the animations more natural
    - Friction
    - -> not perfectly smooth animations
    - -> chaos
    - -> turbulence
- this chapter is about using the 12 principles of animation to make animations more natural and less smooth
  - -> transitions used in the previous chapters have default values which take out the fiction / chaos / turbulence

00:46

Welcome to the real world

② MOVING ON FROM ONLY USING TRANSITIONS IN CSS FOR ANIMATIONS

Now that you're starting to get the hang of animating with transitions, let's take a moment to look at a few things that you can do to create more natural and engaging animations. That isn't to say that there's anything *wrong* with the animations we've created up until now, they're perfectly acceptable. But seeing how we've only been using the most basic of the transition properties, we've unknowingly left a lot of things at their default settings.

- Animating with transitions <- previous chapters
  - Those transitions were done with CSS
  - This chapter is about transition settings
    - -> a transition is an animation
    - -> and you can have settings in that transition -> transition settings
    - -> those settings can e.g add in friction so that there look like there are forces acting on the thing as it transitions <- so the transition is more realistic
      - -> a box colliding against a wall deforming as it collided, rather than being perfectly elastic, e.g
      - -> those effects would be done using the transition properties / settings
- default transitions produce perfect (unrealistic) motion
  - -> in nature there is chaos / friction
  - -> we need imperfect animations, or they will appear robotic
  - -> which is why there are transition properties we can play with
    - They are simulating forces e.g acting on the objects when they transition
    - -> an animation is like a model of reality
      - Transitioning between two states in small integrated increments
      - Doing it within a certain time

And the default settings are ones that produce perfectly smooth, uniform animations. And perfect movement can actually be *too* perfect. You see, nature is entirely too full of chaos to produce perfect movements. We are surrounded by invisible forces, adding noise and aberrations to everything that we do. We're bound by the laws of physics. Objects have mass and must accelerate and decelerate. Movements, on some scale or another, flutter, and dodge, and dive. Our brains interpret irregular motion as natural, so leaving our animations at their defaults comes at the risk of them feeling artificial or robotic.

You add animation to websites to create a more immersive and engaging experience for users, whose brains are programmed to expect imperfections. To create a more engrossing experience, you need to breathe some life into animations.

The nine old men - founding fathers of animation

Through the production of its most iconic films, such as *Snow White* and *Cinderella*, Disney had a core group of animators that became known as "The Nine Old Men," who were responsible for developing the style and techniques that made those movies so iconic.

Two of those nine men, Ollie Johnston and Frank Thomas, wrote a book together titled *The Illusion of Life*, in which they laid out their methods for creating vivid and appealing animation through what they called the "12 Principles of Animation." Although they were framed within the setting of traditional, hand-drawn animation, they've been adapted to nearly every other genre, and are perfectly relevant to creating CSS animations as well.

Let's take a quick look at each of the 12 principles.

Squash and stretch

This principle helps depict an object's mass and momentum, stretching the object's form as it accelerates, and then flattening and widening as it decelerates. Imagine a rubber ball bouncing on a hard floor, flattening out before springing away back up into the air.

Notice how the box stretches horizontally as it accelerates, and squashes from the rapid deceleration as it hits the wall.

Anticipation

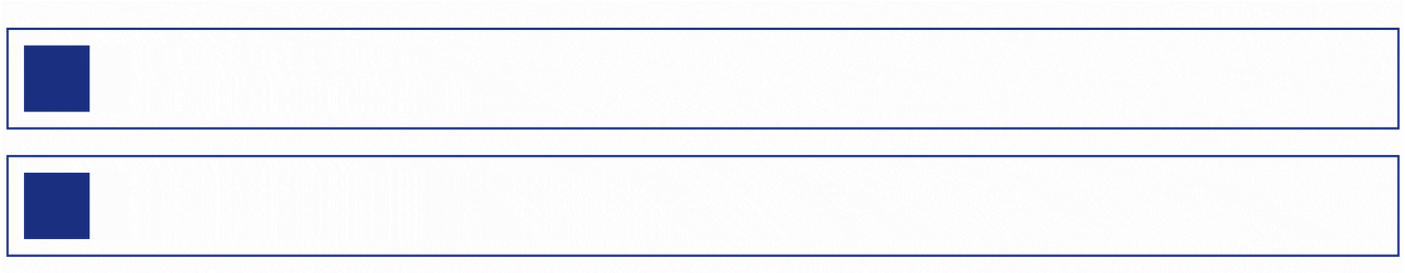
Think of anticipation as a visual foreshadowing of an action to come. A cat coiling and wriggling before pouncing. Cocking an arm back before throwing a ball. Robots move from position to position. Creatures rely on preliminary motion to help gain leverage and power making movement without anticipation seem stale and artificial.

Staging

Staging, with respect to animation, can best be described as composition. It guides the user's eyes to the most important parts of the screen through motion, such as highlighting a button you want them to click on:

Straight ahead and pose-to-pose

Straight ahead and pose-to-pose are two different ways of executing traditional, hand-drawn animation, and don't apply literally to animating for the web. However, within CSS, there are two methods for creating animations: the transitions you've begun to learn about, and keyframes, which we'll cover in a bit. Transitions are based on a starting and ending value that the browser animates between. But with keyframes, you'll learn how to add in stages, or poses:



Follow through and overlapping action

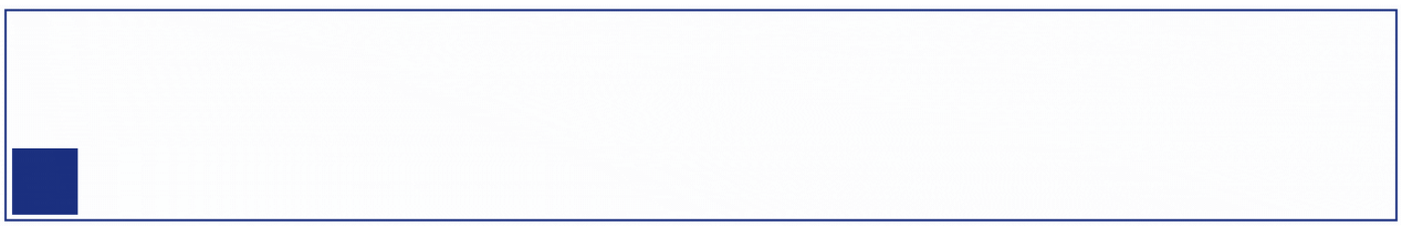
Real objects don't have uniform density and mass, and thus will accelerate and decelerate at different rates. Think of being in a car and having your head snapped back into the headrest as it accelerates. Or having your head bounced forwards if the driver slammed on the brakes. Without follow through or overlapping action, you would remain static within the car as it changed speed. By giving different parts of an object different velocities, animations feel more authentic:

Slow in and slow out

Also known as **ease in** and **ease out,** this principle is based on the fact that physical objects don't simply start and stop instantaneously. They accelerate and decelerate. Slow in refers to the acceleration of an object, and slow out refers to deceleration. Applying different values to eases can change the perception of size or mass of an object. You can make things seem zippier or more cumbersome depending on the profiles of the acceleration and deceleration:

Arc

It's said that nature doesn't make straight lines, and natural movements are no exception. This is a bit trickier with the web's box model, but introducing some arcs into the motion of objects can make them feel more natural:



Secondary action

Adding individual animations to separate pieces of a scene can help to accentuate or strengthen the primary elements of the animation:

Timing

Objects should move at believable rates in comparison to their sizes and mass. A semi-truck is going to accelerate and decelerate more slowly and evenly than a Ferrari, which can get up to speed in the blink of an eye and stop on a dime. You control the timing of objects through their relative sizes, duration of an animation, as well as ease-in and ease-out profiles:

Exaggeration

While trying to keep animations relatable, making them too realistic can result in dry and unappealing motion. Pushing principles of animation a bit beyond the natural limits can help to bestow them with character and personality:

WHAT THE DIFFERENT TRANSITION EFFECTS DO DO THE ANIMATIONS

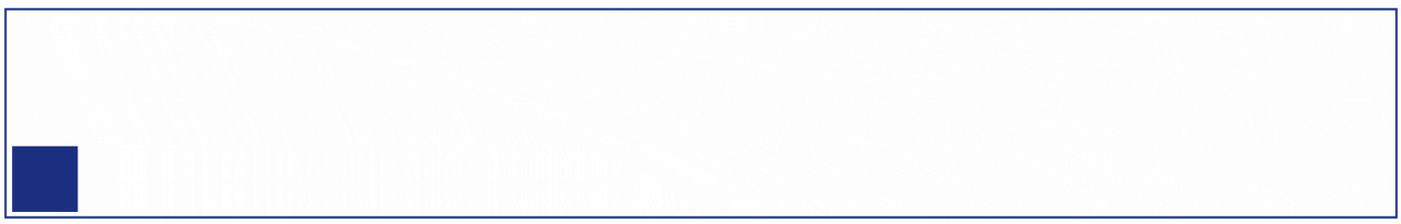
- Animation history
  - -> the nine old men
    - These are the people who developed them
    - -> they developed the effects through Snow White / Cinderella etc <- core animators which made the first Disney films
      - -> they were the ones who developed these techniques
  - -> these 12 principles came from two of those old men
    - -> Ollie Johnston, Frank Thomas
    - -> The Illusion of life book
    - -> “the 12 principles of Animation”
    - -> these were first from hand drawn animation
    - -> they are used for CSS today
- What the different transition properties are doing
  - -> squash and stretch <- when the object is moving, it gets thinner and longer
    - -> basically is stretches
    - -> smash it into a wall and it flattens against the wall
  - -> anticipation
    - -> this is the thing taking off from the wall, rather than smashing into it
    - -> right before it takes off, it gets shorter horizontally <- like a cat jumping
  - -> staging
    - -> for example a line around a button which turns around the button <- you’re taking notes and you circle something on the page
    - -> this is like an animation which circles it, to make it seem more important
    - -> attention economy
  - -> straight ahead and pose-to-pose
    - -> there are stages and poses
    - -> keyframes and transitions
    - -> he’s done an example with a box which expands and then shrinks again as it moves
  - -> follow through and overlapping action
    - -> the lighter the thing is the faster it accelerates
    - -> e.g we have a larger object which is made from two smaller objects. One of those two smaller objects is bigger and one of them is smaller - then the smaller one will accelerate faster
    - -> it’s basically inertia
    - -> key words he’s using are follow through and overlapping action
      - Follow through <- after the force has been applied
      - Overlapping action <- the example above, the larger object being combined with the small one when the inertia is happening
  - -> slow in and slow out
    - -> it starts off moving slow, then is quick in the middle, then slow again at the end
    - -> also known as ease in and ease out
    - -> this makes some objects appear heavier than others
  - -> arc
    - -> e.g a box is moving from one place to another <- introducing a SUVAT parabola
    - -> like throwing it from a to b and there being an arc which it follows and then lands
    - -> rather than just floating from a to b
  - -> secondary action
    - -> a tire rolling and only moving when the forwards force is enough to overcome the friction
    - -> there is a time in between when the wheels are turning and when the car starts to move

- timing <- changing how long the object takes to get from a to b
  - -> the smaller the object, the faster it should take to get from a to b
  - -> the time is takes to get from a to b is a function of its mass
- Exaggeration
  - -> deliberately exaggerating certain parts of an animation in order to make it have personality
  - -> the users know it's not real
  - -> so this is exaggerating facets which make it look unreal -> stretching it longer than it would in reality be stretched e.g
- Solid drawing
  - -> e.g making the path which a box is taking as it's thrown across a field in the air be a triangle, rather than a parabola
  - -> making it follow a shape which is 'solidly' drawn out
    - You could draw out that shape, rather than it coming from Physics
  - -> this one is about perspective / drawing out the animation as you want it
    - -> to make sure you don't end up with the wrong thing
    - -> you need to understand the properties which you're using in css before you use them - or the animation could be incorrect
    - -> it's about the structure of the code
- Appeal
  - -> adding special effects to the animation
  - -> e.g turning a box into a ball before moving it across the screen with an animation -> rather than moving the box across the page
  - -> it's about improving the UX

Solid drawing

Traditionally, this principle refers to properly realizing a scene, such as perspective, etc. It's about accurately structuring animation code. It's critical that you understand the ins and outs of the properties that you are using to ensure that your animations are playing back as desired. If you don't build your code with animations in mind, it can be easy for something like this:

To accidentally turn out like this:



Appeal

In a word: charisma. Adding small flourishes to animations can help make them more dynamic and engaging to users. The life of the party is the life of the party because they attract the attention of the room. It's their appeal and charisma that draws attention to them and keeps it there:

Throughout this course, you'll learn how to harness the power of CSS transition and animation properties to incorporate parts of these 12 principles into your own animations, and create more authentic and engaging experiences for users.

Let's recap!

RECAP

The 12 principles of animation are:

- Squash and stretch
- Anticipation
- Staging
- Straight ahead and pose-to-pose
- Follow through and overlapping action
- Slow in and slow out

- Arc
- Secondary action
- Timing
- Exaggeration
- Solid drawing
- Appeal

Can you define each of them? 🧐

CONTENTS → CREATING MULTI-PROPERTY CSS TRANSITIONS

① VIDEO NOTES
② INTRO / ABOUT THE CHAPTER
③ COMBINING TRANSITIONS
④ SEPARATING TRANSITIONS
⑤ DELAYED TRANSITIONS
⑥ RECAP

Create Modern CSS Animations

6. Create multi-property CSS transitions

Create multi-property CSS transitions

① VIDEO NOTES

- -> using Sass transition properties
  - -> the chapters before the last were about transitions in Sass (indented css)
  - -> the previous chapter was about adding properties to those transitions -> in order to make the animations between the two states look more realistic
    - -> e.g forces / inertia etc
    - -> that was the 12 principles of animation
- -> this chapter is how to code them into the transitions in Sass / css
  - **-> the chapters before the last section were about transitions**
  - **-> this previous one was about how to make them more realistic -> all of the different effects which could be added <- the 12 principles of animation**
  - **-> this is about how to get those animations into Sass**

00:31

Bridging the gap

② INTRO / ABOUT THE CHAPTER

You've just learned about some cool techniques for strengthening and enhancing your animations. But it may seem pretty tough to picture pulling off some of those principles with your transitions though.

You can make things grow and shrink.... but.... how on earth are you ever going to achieve those more organic and interesting animations that you're after?

Complexity is the key to making animations more compelling; they aren't simply moving from point A to point B. Instead, they're layering in other, smaller, bits of motion to create a movement that feels more authentic to the viewer.

That's nice, but that doesn't answer the question. ***HOW?***

Up until now, we've only used the most minimal elements to create transitions. There are more properties that you can add, which will allow you to create more refined and complex animations.

Two for the price of one: combining transitions

The keen observer may have noticed that some of the principles, like **secondary action**, require animating more than one property. I mean, it's right there in the name. **second**ary. But not to worry, you can animate two properties with a single transition, or three, or as many as you want, really.

Previously, we built a button that grows when hovered over. It's built as a `<div>` with the class of `.btn` assigned to it:

```html
<body>
    <div class="container">
        <div class="btn">
            Hover over me!
        </div>
    </div>
</body>
```

Within Sass, we've added a pseudo-selector for the `:hover` state with a transform property that scales it up by 13%, and a .45 second transition:

```scss
.btn {
    background-color: $cd-btn;
    border: 4px solid $cd-btn;
    border-radius: 10rem;
    cursor: pointer;
    font-size: 3rem;
    overflow: hidden;
    padding: 1.85rem 3rem;
    position: relative;
    transition: transform 450ms;
    &:hover {
        transform: scale(1.13);
    }
}
```

Let's spice up the button a bit and layer in some secondary animation by creating a `:hover` state that fades in the background as the button expands. Something like this:

There's a few ways to go about animating opacity, and we'll talk about a more performant (but complex) solution in a bit. But for the sake of simplicity, let's transition the `background-color` between two different `rgba()` values, which we'll define as Sass variables:

```scss
$cd-btn-start: rgba(1, 28, 55, 0);
$cd-btn-end: rgba(1, 28, 55, 1);
```

They both have the same RGB values, but `$cd-btn-start` has an alpha value of 0, making it transparent, whereas `$cd-btn-end` has an alpha value of one, making it opaque, so that the transition will effectively animating the opacity of the color.

Now, to add a transition for our opacity change, we need to add new color variables to the `.btn` selector and its `:hover` state:

```scss
.btn {
    background-color: $cd-btn-start;
    border: 4px solid $cd-btn;
    border-radius: 10rem;
    cursor: pointer;
    font-size: 3rem;
    padding: 1.85rem 3rem;
    &:hover {
        transform: scale(1.13);
        background-color: $cd-btn-end;
    }
}
```

Now we have a button with a functioning `:hover` selector, but we still need to add in the transition properties:

When you want to transition multiple properties at once, rather than typing the name of the property that you want to transition:

```scss
```

```scss
transition: transform 450ms;
```

Instead use the `all` keyword in its place:

```scss
transition: all 450ms;
```

By using the `all` keyword, you are telling the browser to transition all of the properties you have changed within the `:hover` pseudo-selector, rather than a specific property. So let's add a transition to the `.btn` selector to animate both the scale of the button and its background opacity:

```scss
.btn {
    background-color: $cd-btn-start;
    border: 4px solid $cd-btn;
    border-radius: 10rem;
    cursor: pointer;
    font-size: 3rem;
    overflow: hidden;
    padding: 1.85rem 3rem;
    position: relative;
    transition: all 450ms;

    &:hover {
        transform: scale(1.13);
        background-color: $cd-btn-end;
    }
}
```

Now let's check out how our button behaves now:

Wooo! Two things happening at once!

You do your thing, I'll do mine: separating transitions

Using the `all` keyword is perfect for when you want **multiple** things to happen at **the same time,** and for the **same duration,** but there's another way to add multiple properties to a transition. Rather than using `all` , you can list out the animations for each property that you'd like to transition, separated by commas. So, in the case of our transitions for transform and background-color, we could split them like this:

```scss
transition: transform 450ms, background-color 450ms;
```

Now that they're separate, you can give them different durations, which can help break up the uniformity of the animation, creating something more visually interesting to engage the user. Let's make the

`background-color` transition a little faster than the scale by shortening its duration to 300ms:

```scss
transition: transform 450ms, background-color 300ms;
```

Now the `background-color` will be fully opaque before the button has finished scaling up. The difference isn't huge, but those few fractions of a second help make it feel a little less perfect and give it some texture:

By separating the transitions, you can do things to add complexity to your animations, which makes them more interesting and engaging for the user. Notice what I did there? I said **things,** plural. That's right, there's more!

After you... delaying transitions

Right now we are transitioning two properties, with two separate durations, but both begin simultaneously, so the `background-color` is fully opaque before the `transform` transition finishes. What would be nice is if the `background-color` transition waited 150ms before starting so that they both finish their animations at the same time.

We need to use the transition-delay property, which, as you might have guessed, delays the start of transition animation by however much time assigned as its value. To delay a transition by 150ms, write the code like this:

```scss
transition-delay: 150ms;
```

Since there are two transitions occurring, you need to specify a value for each, in the order that they are declared in the transition property. In our case, we don't want to delay the transform animation, while we delay the background-color animation by 150ms:

```scss
transition: transform 450ms, background-color 300ms;
transition-delay: 0, 150ms;
```

Now the transitions will start 150ms apart and finish simultaneously. But, just like `transition-property` and `transition-duration`, you can write the same thing more concisely by adding `transition-delay` values to the transition property itself, after the `transition-duration` values:

```scss
transition: transform 450ms, background-color 300ms 150ms;
```

In the case of an absence for a delay value, the `transition` property will assume a zero delay, so you can omit it from the `transform` transition. Let's take a look at our transition within the context of its Sass block:

```scss
.btn {
    background-color: $cd-btn-start;
    border: 4px solid $cd-btn;
    border-radius: 10rem;
    cursor: pointer;
    font-size: 3rem;
    overflow: hidden;
    padding: 1.85rem 3rem;
    position: relative;
    transition: transform 450ms, background-color 300ms 150ms;
    &:hover {
        transform: scale(1.13);
        background-color: $cd-btn-end;
    }
}
```

And here is the end result for our button:

Hover over me!

Things in life don't happen simultaneously. If you were to watch ultra-slow-motion footage, you would see that things happen as part of a cascade. One thing leads to another, and another. And our brains have come to expect there to be separations between events, even if they are only fractions of a second.

By creating differences in the timings of animations, both through the durations as well as offsetting their starts through delays, you can create more authentic, interesting, and entertaining visuals.

Coming up next, we'll integrate slow-in and slow-out, the sixth principle of animation, into our transitions, using the `transition-timing-function` property to manipulate the velocity of animations over the course of their durations.

Let's recap!

- Transitions can contain animations for as many properties as you'd like.
- Use the `all` keyword to transition all properties simultaneously.
- Separate the animations with commas, which allows you to set different values for each.
- Offset the start of transitions by using delays.

Create Modern CSS Animations

CONTENTS FOR USING TIMING FUNCTIONS FOR ANIMATIONS

① VIDEO NOTES ② LOCK AND LOADED ③ RECAP
② EASE IN / EASE OUT TRANSITIONS ④ ROLL YOUR OWN

Use timing functions to create more natural animations ① VIDEO NOTES

- Every object which moves starts and stops
  - -> this involves accelerating and decelerating
  - -> the rate that the velocity is increasing
  - -> it's about the mass of the object
- -> this is using distance time graphs for objects being animated in CSS
  - -> the gradient is the velocity
  - -> the rate of change of the gradient is the acceleration
  - -> it's about the Physics of animations -> of objects being animated
- -> those curves are used in CSS and this is how
  - -> they are automatically there -> the transitions
  - -> you can set them, and your own
- -> adding the curves into transitions and making your own for the transitions in CSS

00:52

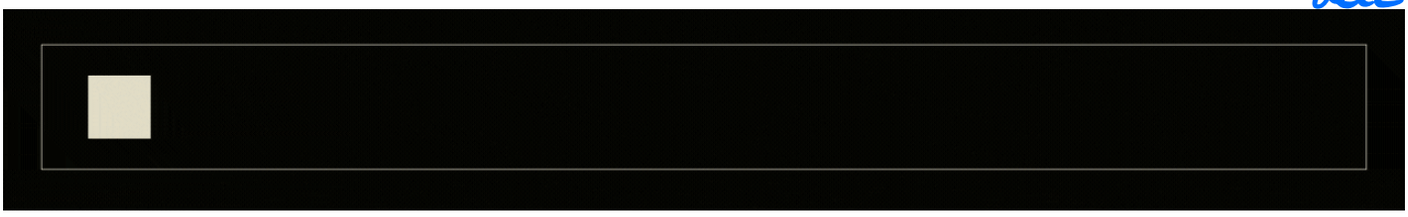
Easy come, easy go ② EASE IN / EASE OUT TRANSITIONS

Every object that has ever moved and ever will move does **two** specific things. To begin moving, they have to **accelerate,** and before they can stop, they have to **decelerate.**

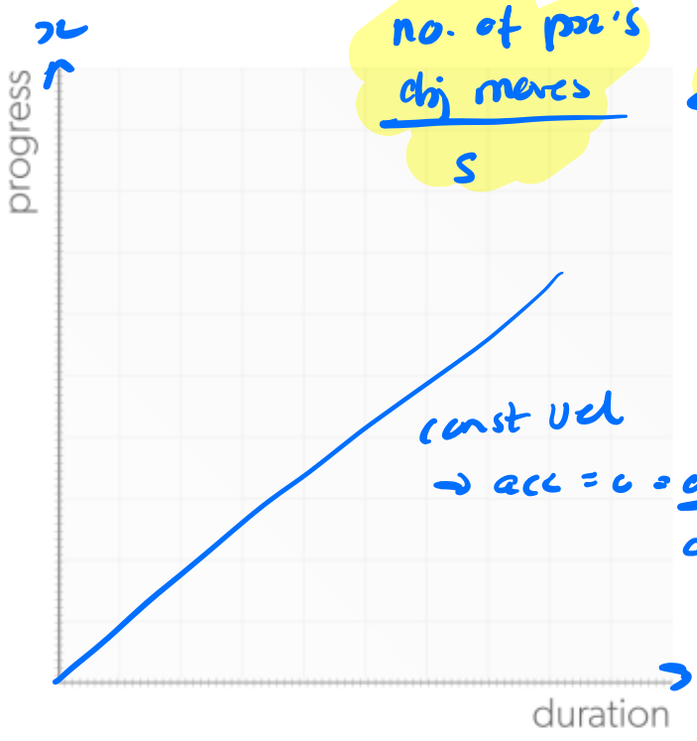
A bullet fired from a gun accelerates more rapidly than a thrown ball, but they both accelerate and it's those rates of acceleration that our brains use to estimate an object's mass and momentum, not to mention where it's going to be and when.

If you want your animations to feel grounded and authentic, then it's vital that you pay attention to how objects accelerate and decelerate, or, in animation terminology, how they **ease in** and **ease out.**

Let's say that we want to animate a box to move 1000 pixels in 1000 milliseconds:



Let's talk about how we'd like our box to accelerate and decelerate, but since we're talking about a highly visual topic, a picture can be worth a thousand words. So let's break down an animation's velocity on a graph:

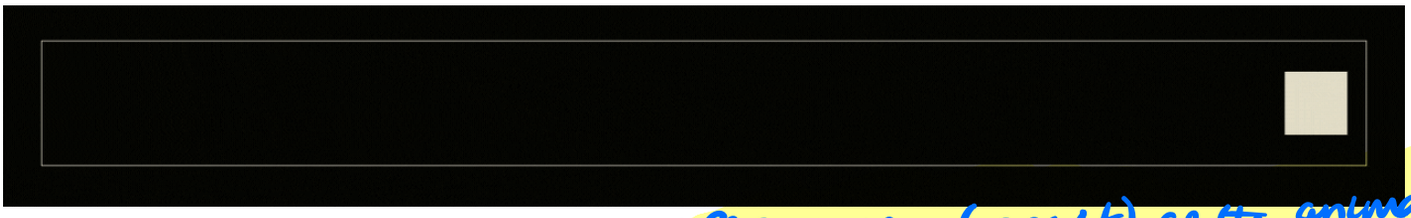


The X-axis represents the percentage of the duration completed, and the Y-axis represents the percentage of the motion completed. Start at 0% duration and 0% progress in the lower left corner, and finish in the upper right corner, where both the progress and durations have reached 100%.
With the starting and ending points, you can connect them with a path that represents the velocity of your animation:

Notice that the velocity isn't a straight line. Instead, its slope changes as the box accelerates at the beginning and decelerates at the end. In the beginning, the box isn't moving very quickly yet, so it hasn't completed as much progress as it has duration, making the line flatter. At around 20% into the duration-axis, the box has gotten up to speed, and it is completing its progress at a faster rate, so the line gets steeper.

But then the box needs to hit the brakes and begin decelerating, reducing the rate of progress, allowing the amount of duration completed to overtake the progress and flatten the slope of the curve as it approaches the finish. When you look at the graph, you can see the velocity of the animation at any point by the steepness of the curve. The steeper the slope, the higher the velocity; the flatter the slower.

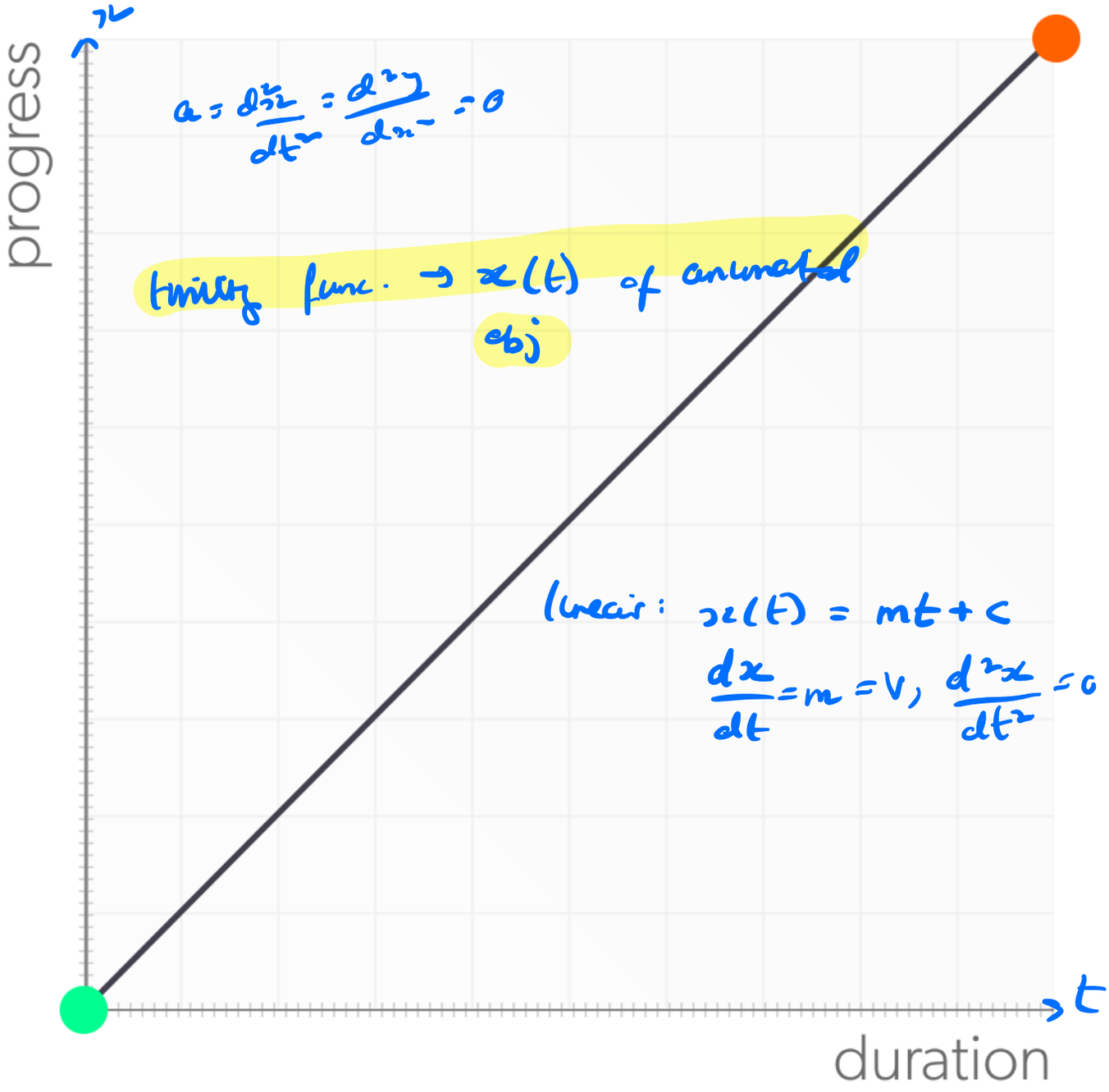
Let's look at the box again, keeping in mind its acceleration curve:



Notice how it gets up to speed pretty quickly out of the gate, but slows down more gradually? In CSS, these acceleration curves are called **timing functions**, and you can use the `transition-timing-function` property to set the acceleration curves for your transitions.

Locked and loaded

CSS comes preloaded with a variety of its own curves right at your fingertips. Let's begin with the curve that wasn't. A `linear` timing function has no acceleration or deceleration to its velocity. Instead, the value goes straight from point A to point B:



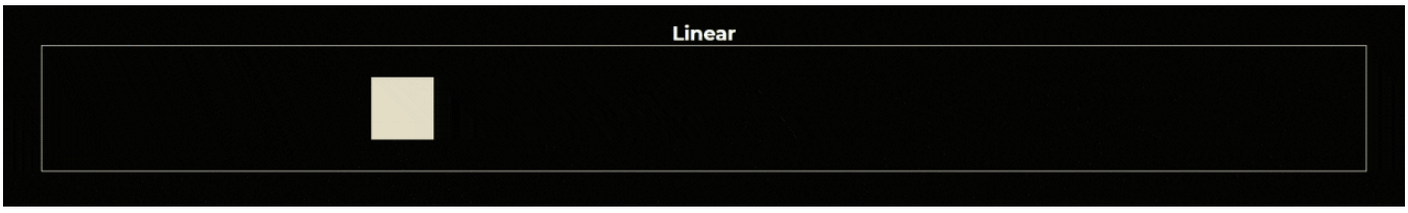
To apply a linear timing function to our box animation, we add the `transition-timing-function` property and set its value as `linear` :

```scss
transition: transform 1000ms;
transition-timing-function: linear;
```

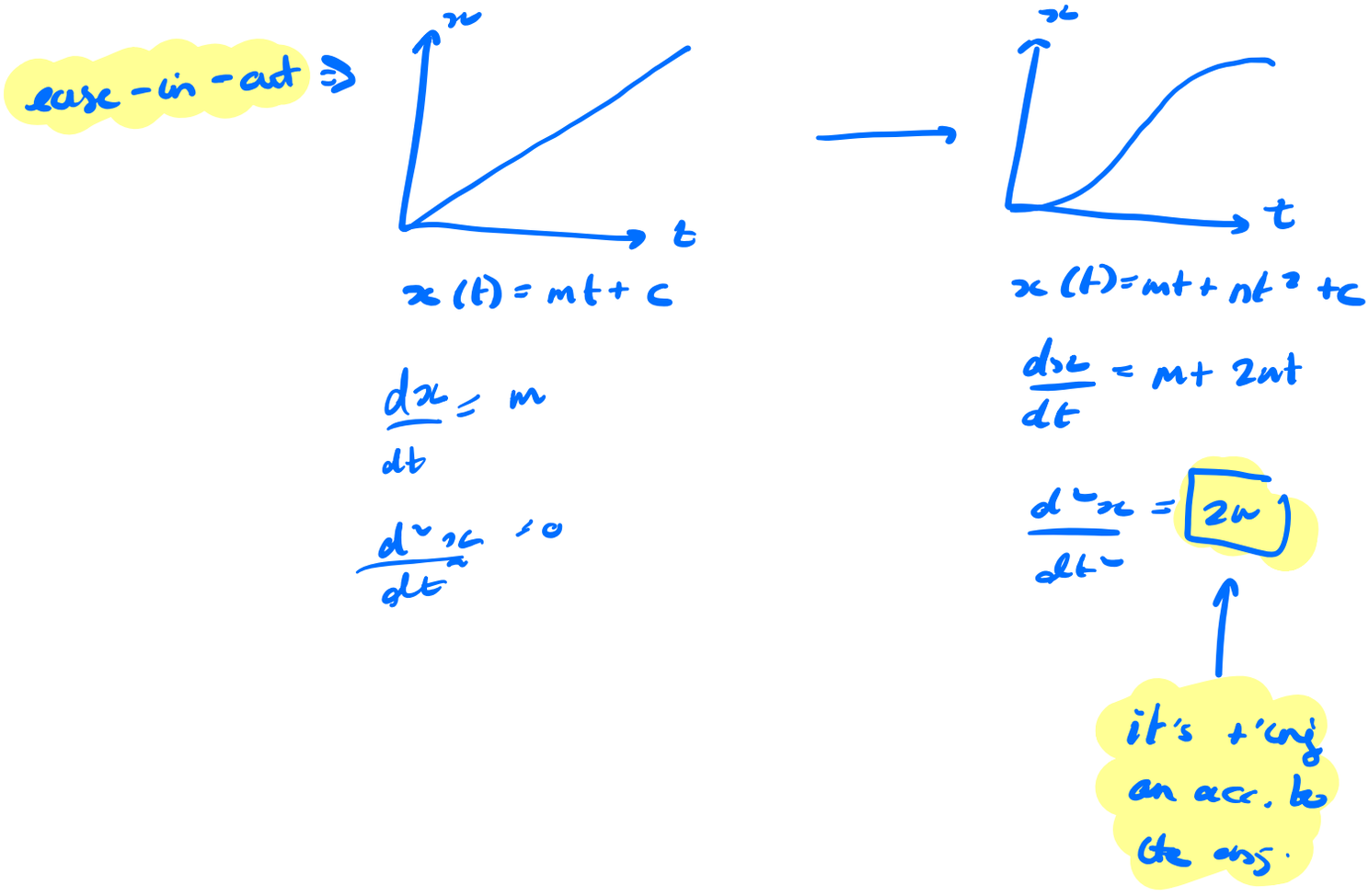
Or add it to the transition property's shorthand by including the linear keyword in its value list:

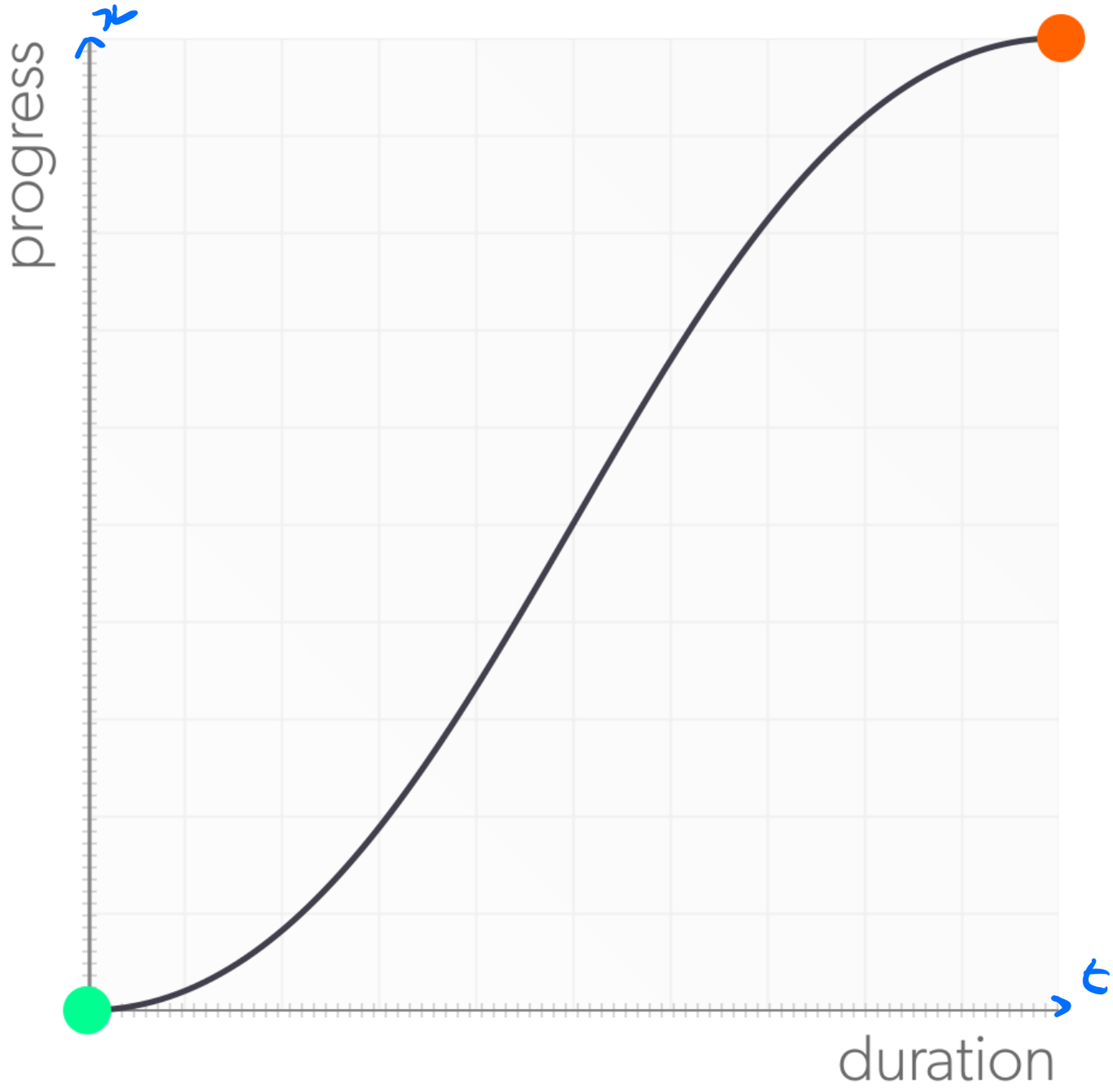
```scss
transition: transform 1000ms linear;
```

And when you check out our box in motion with a linear acceleration curve, you see that it's traveling at a constant speed from start to finish:



Next in the arsenal of built-in timing functions is `ease-in-out` . This is many animator's go-to timing function because it has subtle acceleration and deceleration curves, which you can see in its graph:



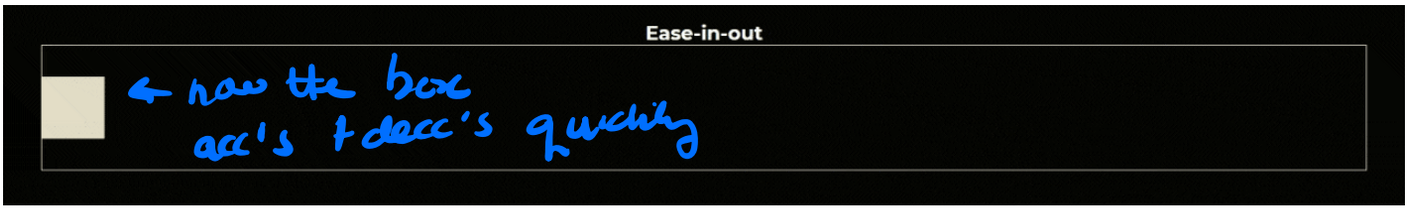


Let's swap `ease-in-out` into our box transition:

scss

```scss
transition: transform 1000ms ease-in-out;
```

And take a look at it in motion:



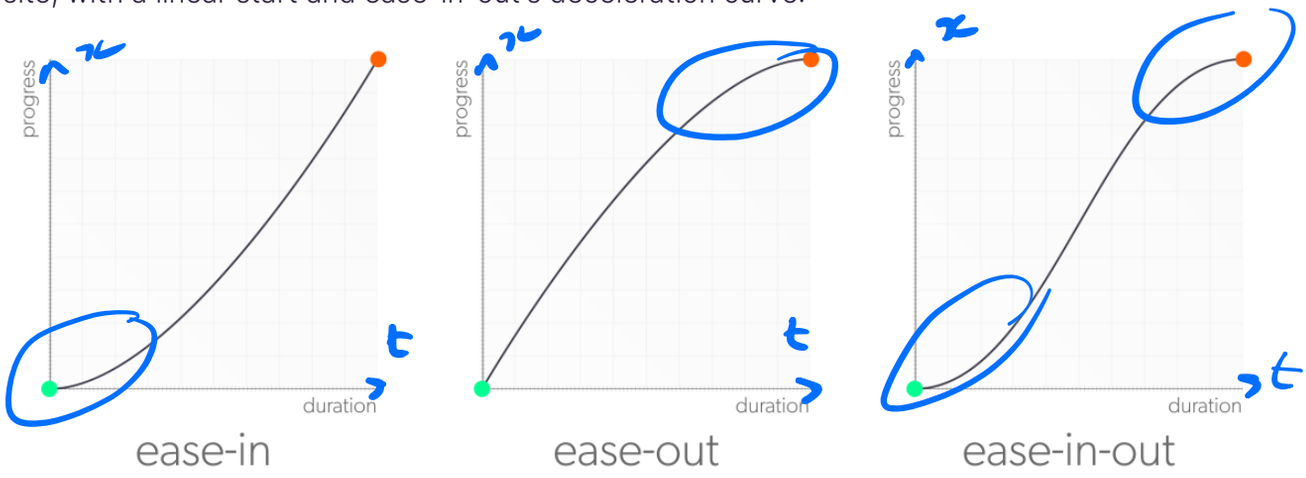
It might take a moment or two or staring at it, but eventually you can see speed ramping up and down, but it's subtle enough to sort of disappear into the motion. In other words, it looks and feels *natural*. And natural is good!

You also have the option to just ease in, or ease out of motion with the **ease-in** and **ease-out** keywords, respectively.

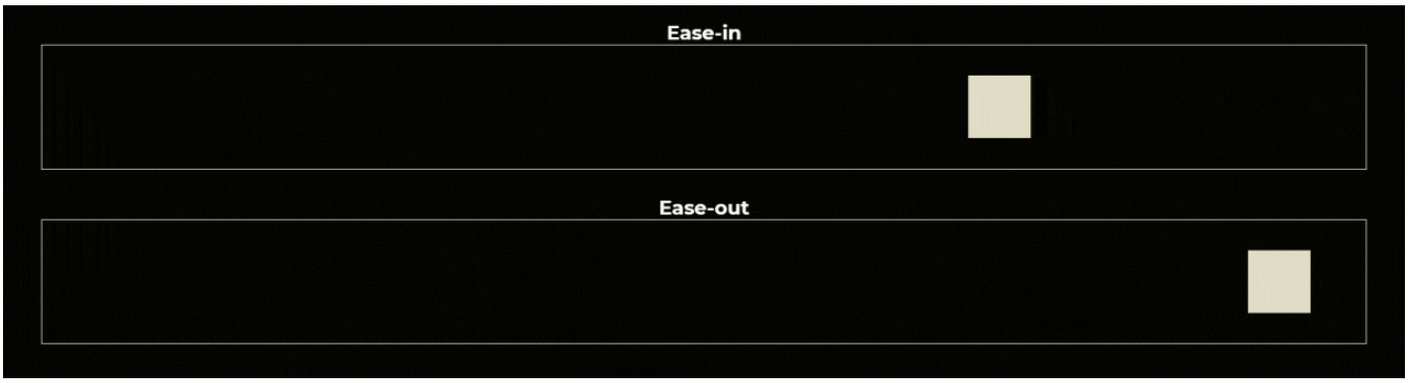
```scss
// ease-in timing-function
transition: transform 1000ms ease-in;


// ease-out timing-function:
transition: transform 1000ms ease-out;
```

When you compare their curves to the `ease-in-out` graph, you see that the `ease-in` graph is essentially the start of the `ease-in-out` graph, with a linear ending. And `ease-out` is just the opposite, with a linear start and ease-in-out's deceleration curve:

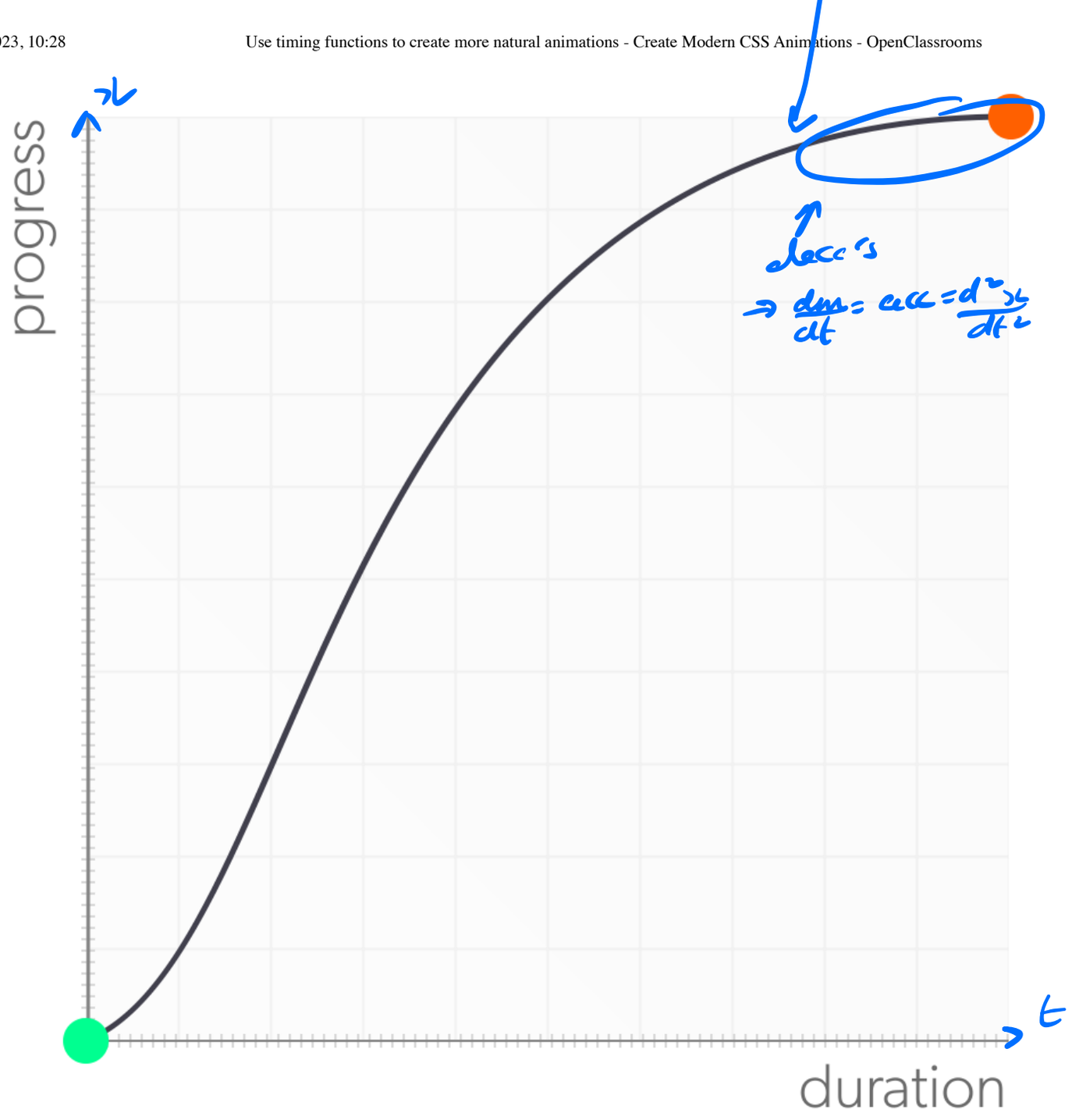


And when you check them out in action, you can see that ease in is slower out of the gate, while ease out hits the brakes earlier:



When you see the two functions side by side, it becomes apparent how much the timing-function of the animation influences the perception of motion. They both take exactly the same amount of time to complete the trip, but thanks to their different curves, the motion, and even duration, don't feel the same.

And for all of the transitions, we've made so far? What kind of timing function is being applied to them? If you don't manually choose one, the browser will apply the `ease` timing function, which sounds a lot like the other curves we've just covered. However, it has its own distinct profile, featuring a sharper acceleration profile, while the deceleration ramp is more pronounced:



If `ease` 's curve looks a bit familiar, that's because it is! It's the curve our box was using at the start of this chapter. Let's take a look at it in context with the rest of CSS's built-in timing functions:

Roll your own

ROLL YOUR OWN

CSS' built-in timing functions give more flexibility with the velocity of transitions, but sometimes you might want a little more control over your acceleration curves. Think back to the lessons you learned from our ball-growing button. Hey, this is serious stuff. No jokes... We've taken our ball-growing rig from earlier and transformed it into a strength-o-meter:

→ choosing the right clone

When you click and hold on the button, the ball shoots to the top of the meter. It works fine, except... it lacks any sort of tension or suspense. There's never any doubt that the ball will reach the top, so there's

nothing to really engage the user.

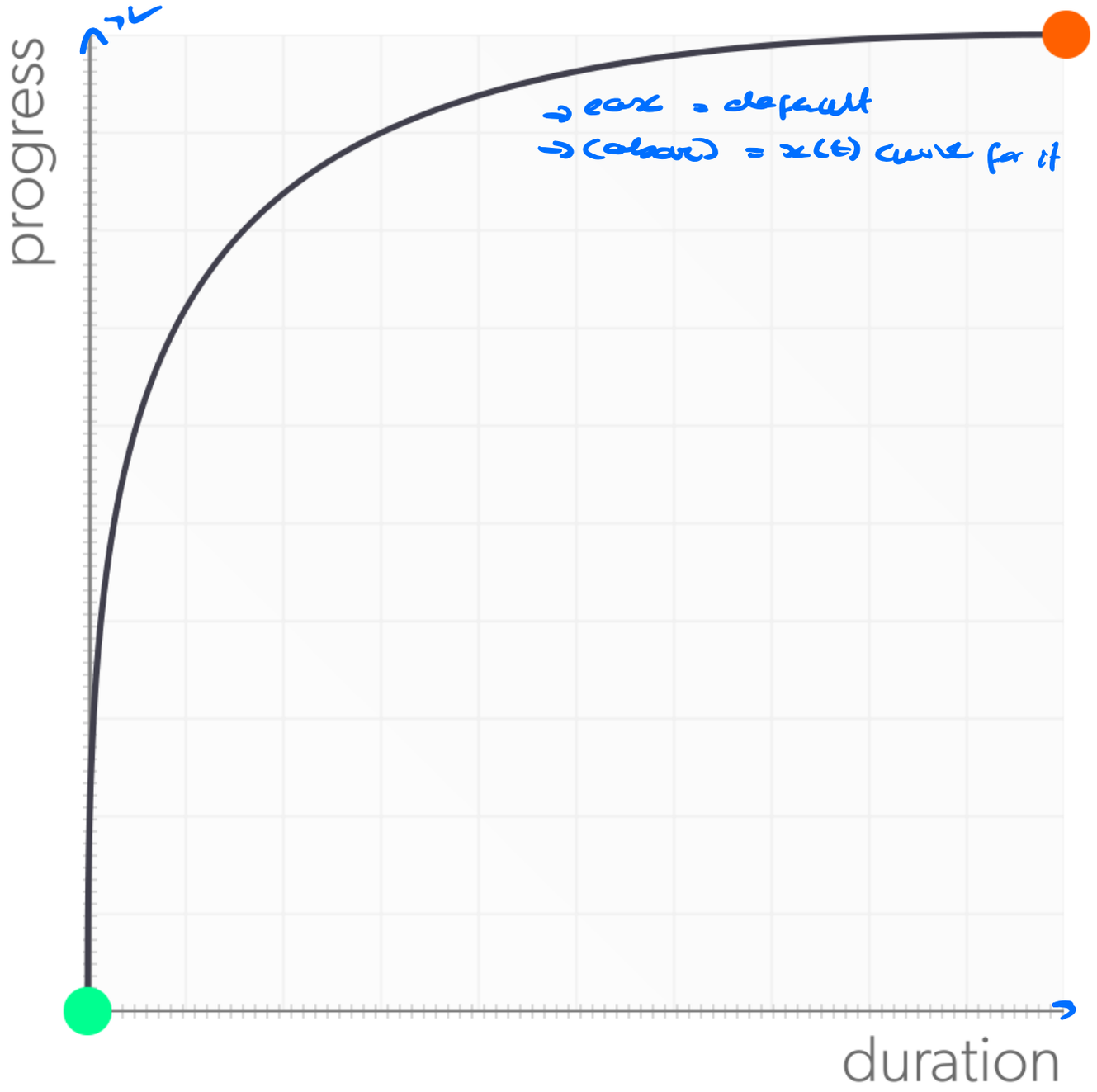
What could help is to change the velocity of the ball, so that it feels like less of a sure thing. Right now the transition is the default `ease` timing function:

scss

```scss
$trans-dur: 2000ms;

transform: translateY(100%);
transition: transform $trans-dur;
```

`ease` has a slight ease in and a bit stronger ease out. If you made the ball shoot like a rocket, only to go slower and slower as it approaches the top, then getting the ball all the way up becomes a feat of patience, encouraging users to mash their mouse buttons, willing it to reach the top. To plot out this acceleration on a curve, it might look something like this:



That curve doesn't look remotely like any of the presets we've covered... and that's because if you want the ball to follow this curve, you'll have to make it yourself. CSS' built-in timing functions are really just shorthand for **cubic-bezier** functions.

The cubic-bezier function is just like the `rgb()` function, where you enter a list of numerical values, but rather than turning those numbers into a color, the cubic-bezier function turns them into an acceleration curve. For example, `ease-in-out` can be written as:

```css
.selector {
    transition-timing-function: cubic-bezier(.42, 0, .58, 1);
}
```

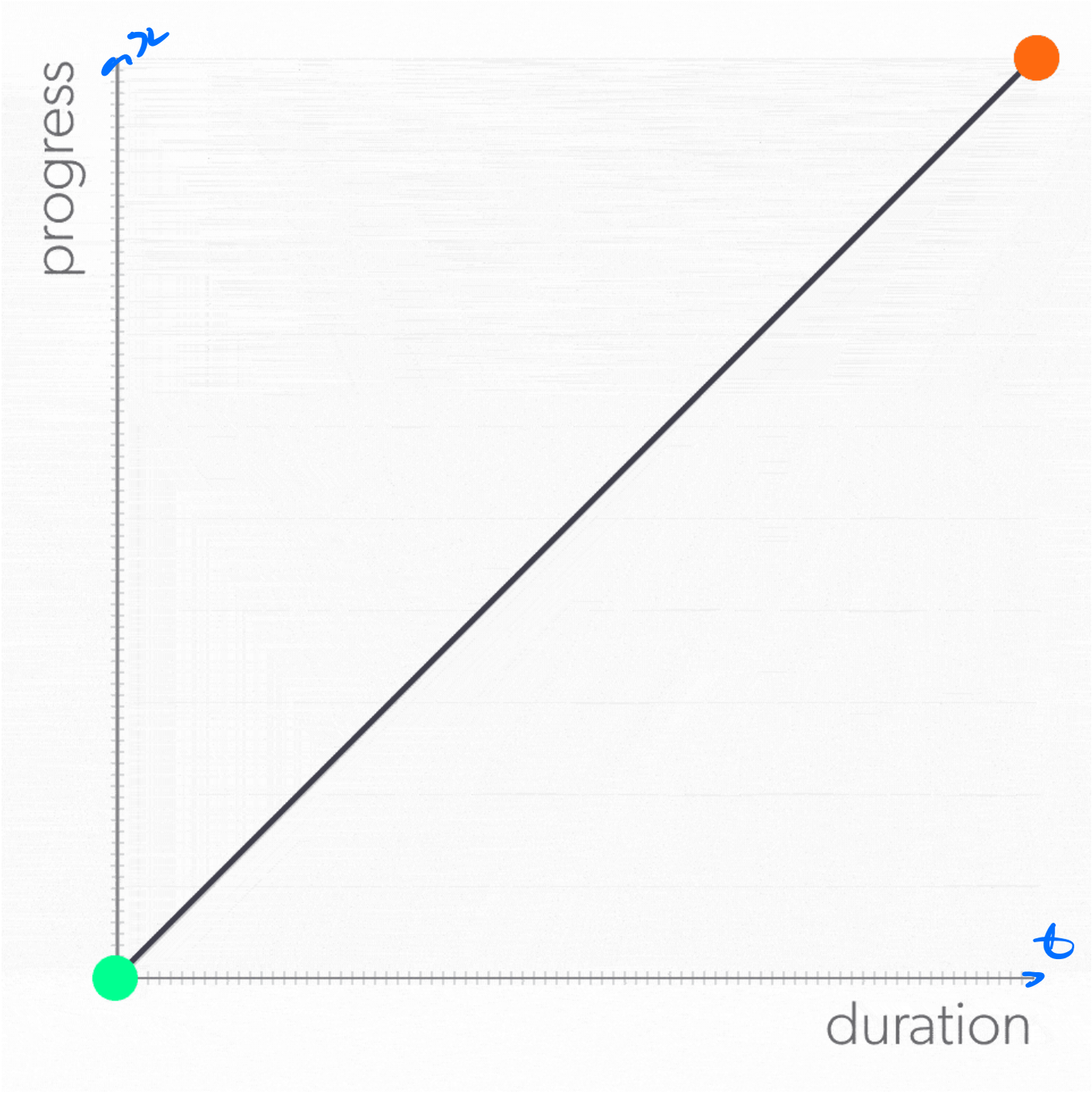
Cool! But... how do you turn an acceleration curve into a `cubic-bezier()` function?

`cubic-bezier()` requires a list of four numbers as arguments, which are actually the coordinates for two points that live on the plane of the graph. The first two numbers are the X and Y coordinates for the point that determines the ease in of the acceleration curves, and the second pair determines its ease out:

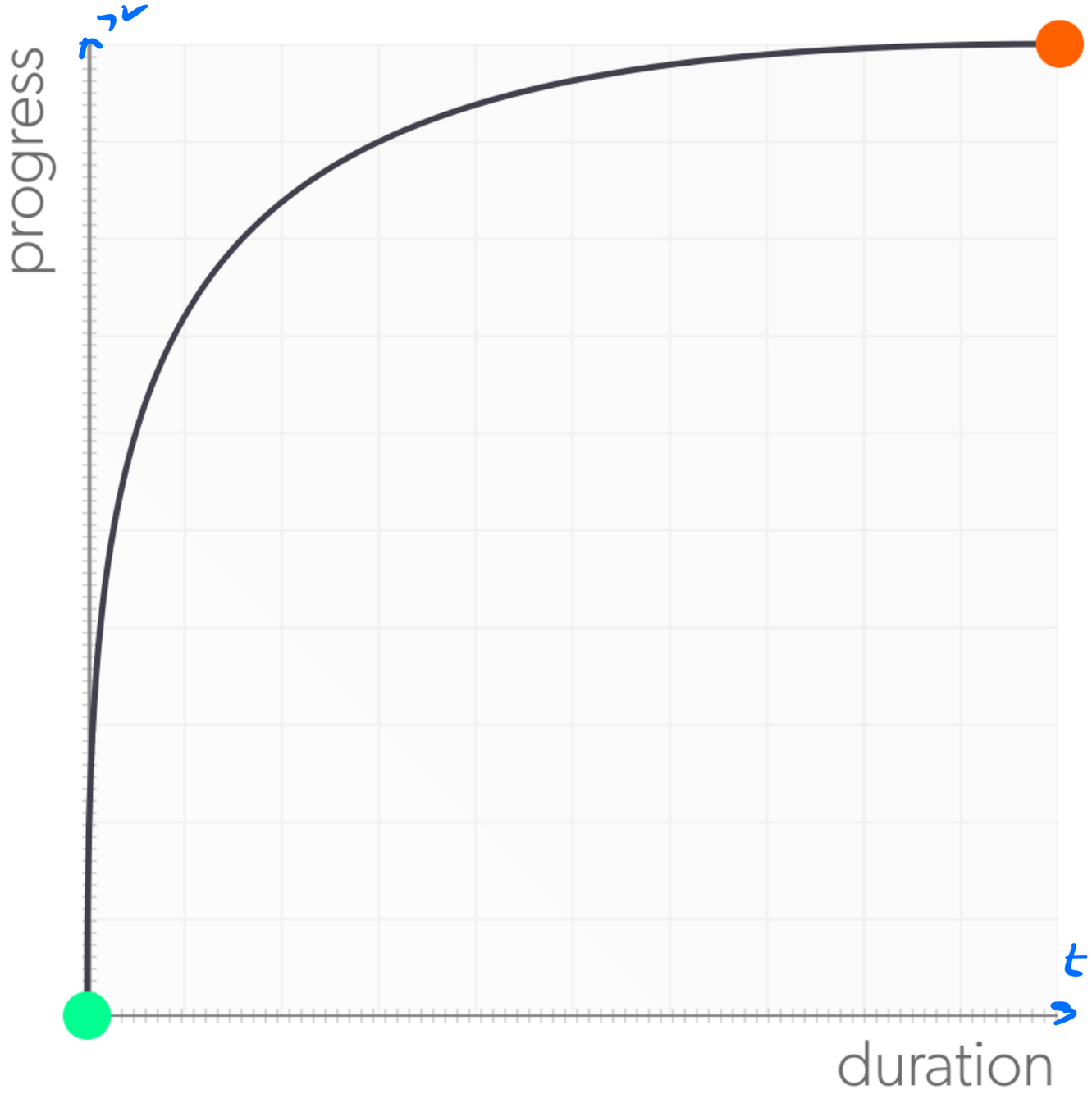
```css
.selector {
                                        /*(X1,Y1)(X2,Y2)*/
    transition-timing-function: cubic-bezier(.42, 0, .58, 1);
}
```

Now, imagine that an acceleration curve is a rope, tied taut between two trees. If you grabbed that rope in two different spots and pulled in two different directions, you would force the rope to deflect from its line in those directions. The coordinates that you enter into `cubic-bezier ()` represent the force and direction that you want to the deflect the line of your acceleration curve.

Let's plot the coordinates of `ease-in-out` onto the graph and look at how they create its acceleration curve:



Neat! ...Buuuuut... *how* do we turn this:

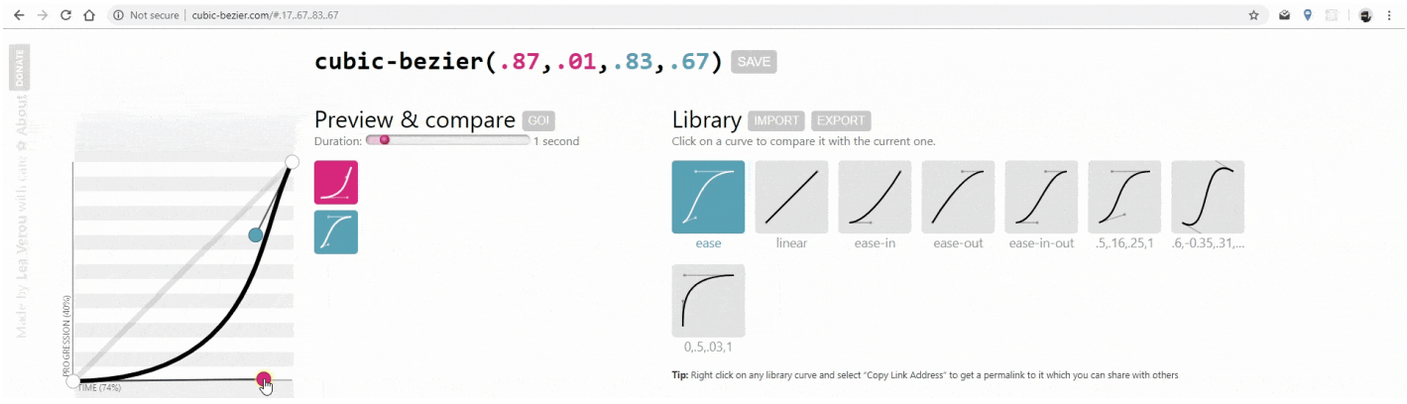


Into this: `cubic-bezier(0, .75, .08, 1)` ???

You could just take a guess at the coordinates and go from there. Or, you could use an online tool to help build it, such as the aptly named cubic-bezier.com.

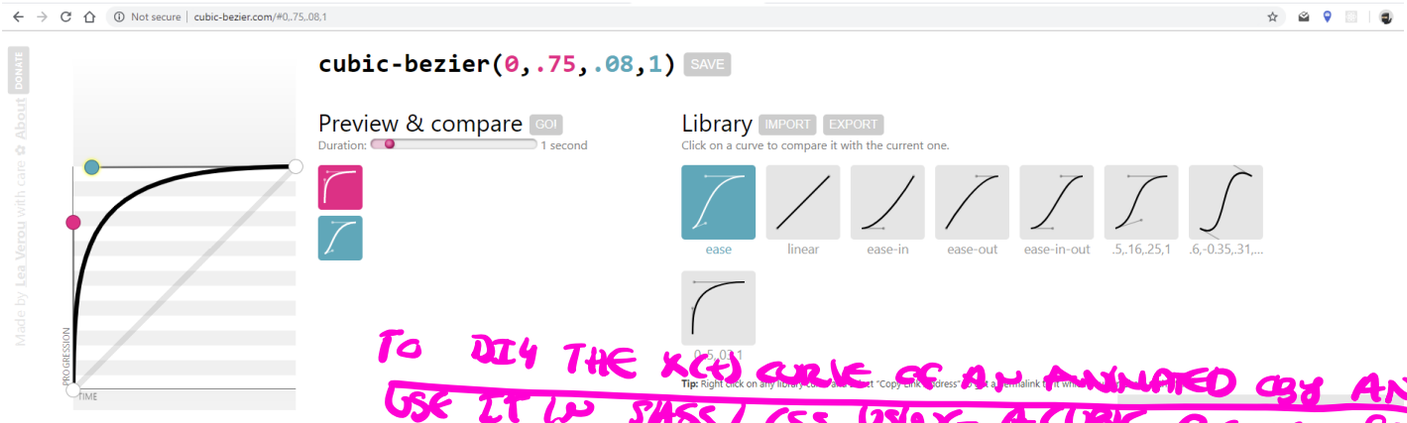
> We'll be using cubic-bezier.com, but there are a ton of other sites out there that you can use as well. Just search for "cubic bezier."

When the site loads, you'll see a graph to the left of the page with a preloaded curve, along with a pair of handles that you can drag around to manipulate the acceleration profile:



Notice how the values of the cubic-bezier function update as the handles are dragged around? To create a custom curve, drag the ease-in and ease-out handles until you've created a curve that you like, then use the generated cubic-bezier values in your code base.

Here is the curve that I've generated for our strength-o-meter:



TO DO THE X(t) CURVE OF AN ANIMATED OBJ AND USE IT IN SASS / CSS, USING A CUBIC BEZIER FUNC.

Let's plug those `cubic-bezier()` values into our transition and take it for a spin!

scss

```scss
$trans-dur: 2000ms;

.strength {
    transform: translateY(100%);
    transition: transform $trans-dur cubic-bezier(0, .75, .08, 1);
}
```

2s

a) he gets a x(t) curve according to how he wants it using the cubic bezier tool - he picks the shape of it, it gives the eq.

b) he plugs the eq. of that curve into Sass

Now our transition has more tension! The ball shoots out of the gate, but creeps into its finish:

code for the animation

transition: transform 2s cubic-bezier(... ,... ,... ,...)

it moves from a to b

t for animated obj. to transition

x(t) curve of animated obj.

The assigned timing function not only helps to make them seem more natural, but they can also be used to tell a story. And while one of the presets may work well a lot of the time, sometimes a custom curve is necessary to create the desired effect.

Let's recap!

- The acceleration and deceleration of transitions is controlled using the `transition-timing-function` property.
- If no timing function is declared, the transition will default to using the `ease` function.
- Other keyword timing functions include `ease-in`, `ease-out`, `ease-in-out`, and `linear`.
- When a preset timing function doesn't fit the animation, custom animation curves can be declared using the `cubic-bezier()` function.

Max Wardeh

Have you built a strong foundation in CSS animations?

Evaluated skills

- Apply the 12 Principles of Animation
- Implement simple CSS transitions

Question 1

You have a button with a class of `.btn`, and you want to animate it by increasing its size by 20%, and its background color to change from orange to green when hovered over. Which of the following (written in Sass) would achieve this?

```css
.btn {
  background-color: orange;
  transform: scale(1);
  transition: scale, background-color 600ms;
  &:hover {
    background-color: green;
    transform: scale(1.2);

  }
}
```

css

```css
.btn {
  background-color: orange;
  transform: scale(1);
  transition: all 600ms;
  &:hover {
    background-color: green;
    transform: scale(20%);

  }
}
```

css

```css
.btn {
  background-color: orange;
  transform: scale(1);
  transition: scale 600ms;
  &:hover {
    background-color: green;
    transform: scale(1.2);

  }
}
```

css

```css
.btn {
  background-color: orange;
  transform: scale(1);
  transition: all 600ms;
  &:hover {
    background-color: green;
    transform: scale(1.2);

  }
}
```

the first option is better, because it's explicitly listing out the effects we want to hgt.

Question 2

Which of the following are among the essential ingredients for a CSS animation transition?

Careful, there are several correct answers.

Careful, there are several correct answers.

- [x] A property to animate.
- [x] The duration of the transition.

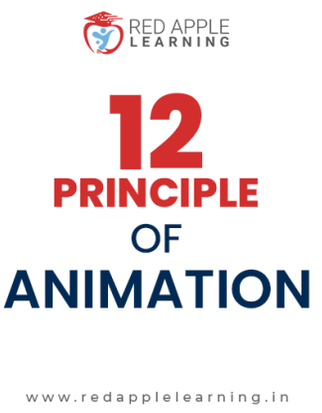
- [ ] The end time of the transition.
- [x] A pseudo-selector to trigger the animation.

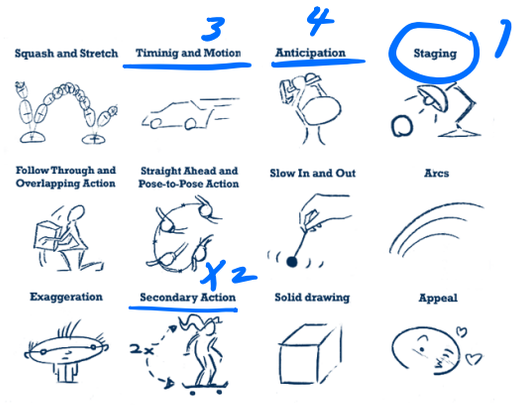
Question 3

Which of the following are among the 12 principles of animation?

Careful, there are several correct answers

- [x] Staging
- [ ] Primary action
- [x] Timing
- [x] Anticipation





Question 4

You want to create an animation where a rocket is launched by clicking a button on the page:

Which of the following HTML and CSS will achieve this?

HTML:

```html
<div class="container">
    <div class="btn">
      Launch!
    </div>
    <div class="rocket"></div>
</div>
```

← the two are next to each other

CSS:

```css
.btn {
  &:active + .rocket{
      transform: translateY(0%);
  }
}

.rocket {
  transform: translateY(100%);
  transition: transform 4s;
}
```

← if one element being e.g. hovered over is affecting a transition in another el., then the linked el. has to be its adjacent sibling in html

```html
<div class="container">
    <div class="btn">
      Launch!
      <div class="rocket"></div>
    </div>
</div>
```

```css
.btn:active .rocket{
  transform: translateY(0%);
}

.rocket {
  transform: translateY(100%);
  transition: transform 4s;
}
```

✗ should be + → the pseudoclass of one el should be affecting the other

```html
<div class="container">
    <div class="rocket"></div>
    <div class="btn">
      Launch!
    </div>
</div>
```

← should be swapped so this works

```css
```

```css
.btn {
  &:hover + .rocket{
      transform: translateY(0%);
  }
}

.rocket {
  transform: translateY(100%);
  transition: transform 4s;
}
```

html

```html
<div class="container">
    <div class="rocket"></div>
    <div class="btn">
      Launch!
    </div>
</div>
```

css

```css
.btn {
  &:active + .rocket{
      transform: translateY(0%);
  }
}

.rocket {
  transform: translateY(100%);
  transition: transform 4s;
}
```

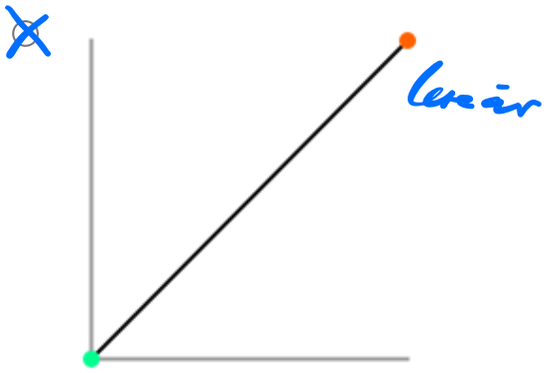
Question 5

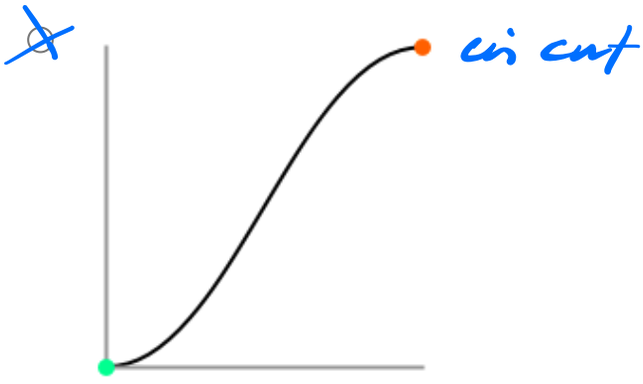
Which is many animators' go-to timing function because of its subtle acceleration and deceleration?

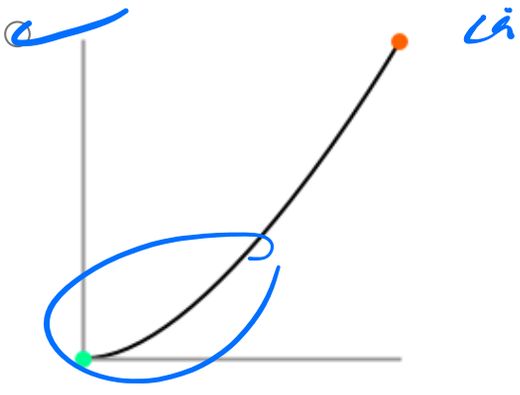
- Linear
- Ease-in-out
- Ease-out-in
- Ease-out

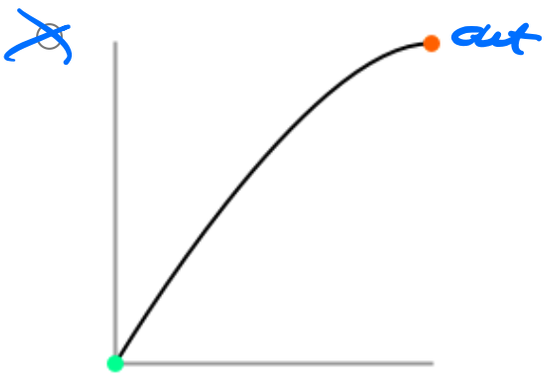
Question 6

Which of the following curves represent an ease-in timing function?









Question 7

Which of the principals of animation does the transition-timing-function CSS property help apply to an element?

diy x(t) curves for objects being animated → cubic bezier func.

Careful, there are several correct answers.

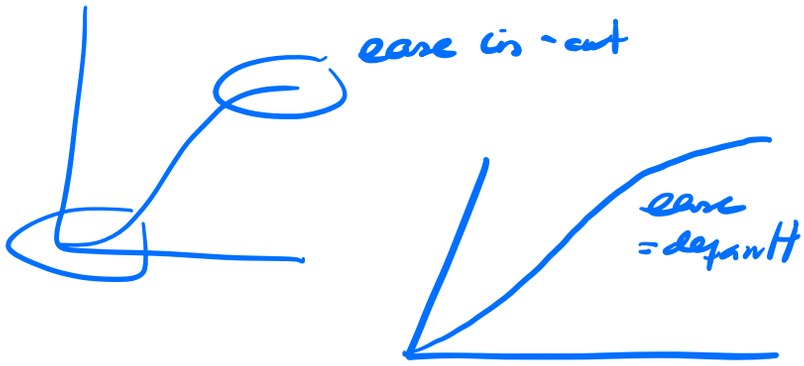
- [x] Anticipation
- [ ] Timing ← the x(t) graph covers the same t

- [ ] Straight ahead and pose-to-pose
- [x] Slow in and slow out

Question 8

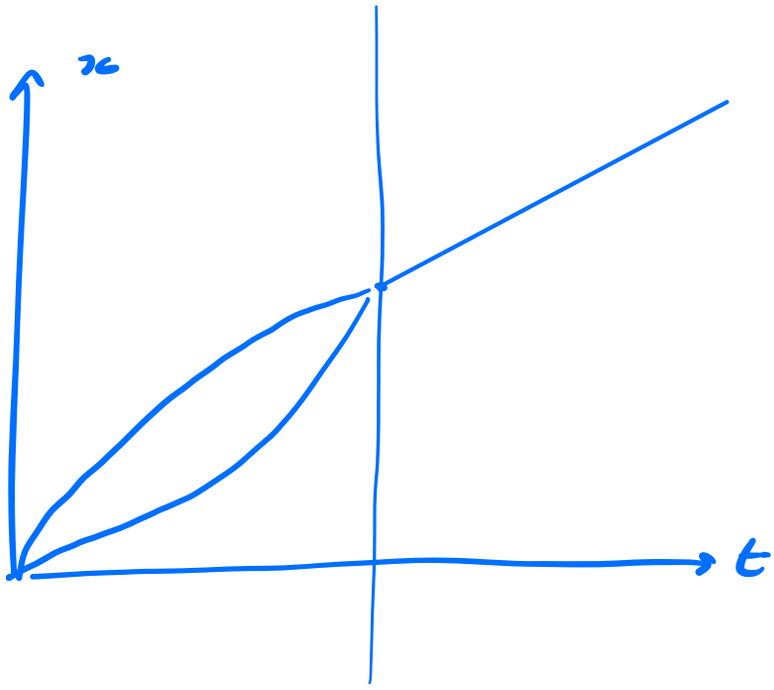
Since the animation in question four does not have the transition-timing-function set, the default acceleration curve is being applied. Which one is the default?

- ○ Ease-in
- ◉ Ease
- ○ Ease-in-out
- ○ Ease-out



Question 9

You want the rocket launch animation in question four to be more realistic by applying a custom acceleration curve using the `cubic-bezier()` function. You want it to appear to struggle to get off the ground—as it's fighting gravity—when the animation starts. It should then travel at a steady speed upwards when it reaches about halfway up. Like this:



This is what the acceleration curve looks like:

Which of the following functions describes this acceleration curve?

Hint: you can use a tool such as cubic-bezier.com to find the parameters

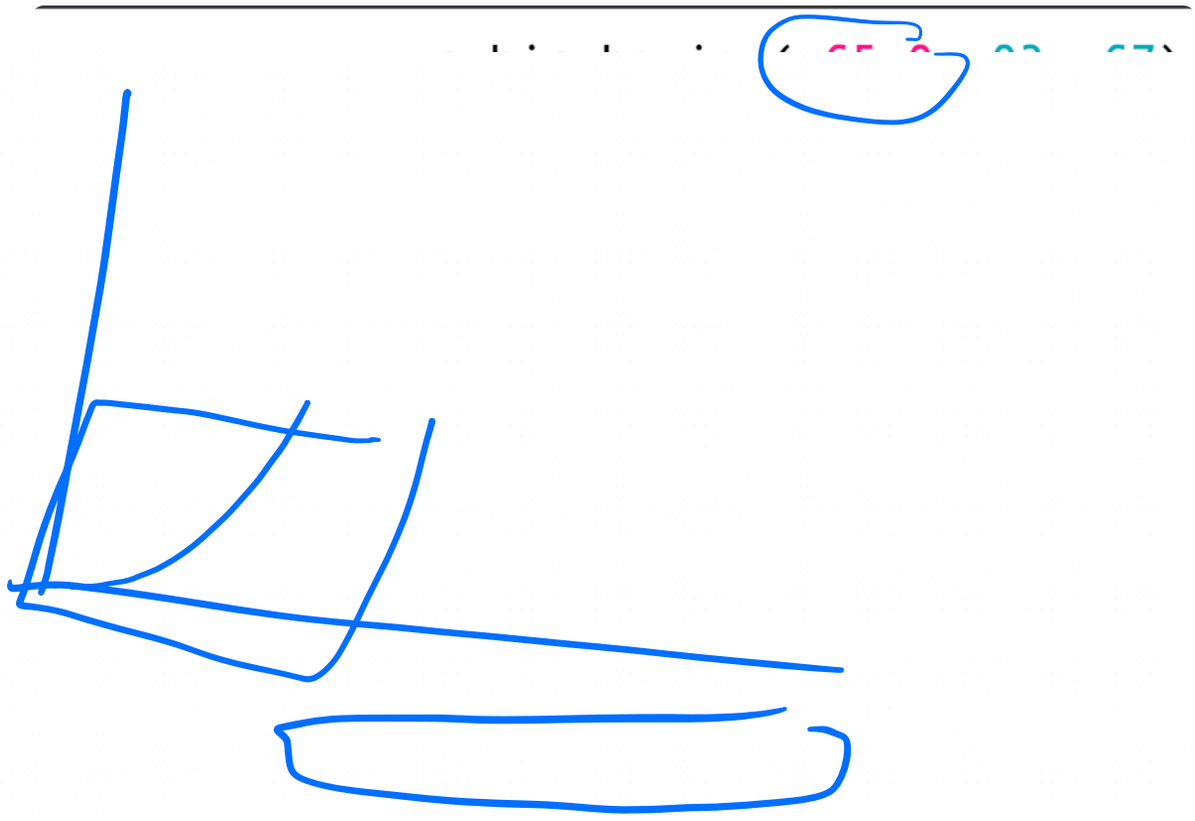
- ○
```css
cubic-bezier(.65,0,.28,.79)
```
- ○
```css
cubic-bezier(0,.65,.28,.79)
```
- ◉
```css
cubic-bezier(.65,0,.79,.28)
```

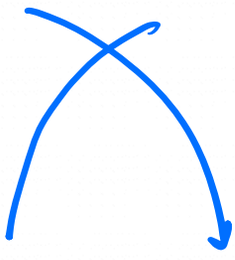
→ put them into a graphing calculator
- there are online calc. tools
for the eq. of the curve of the rocket

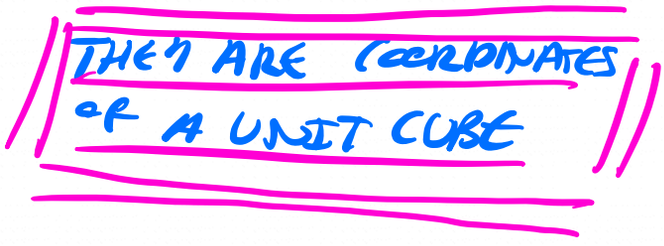
○

```css
cubic-bezier(0,.65,.79,.28)
```

You need to answer 9 more questions.

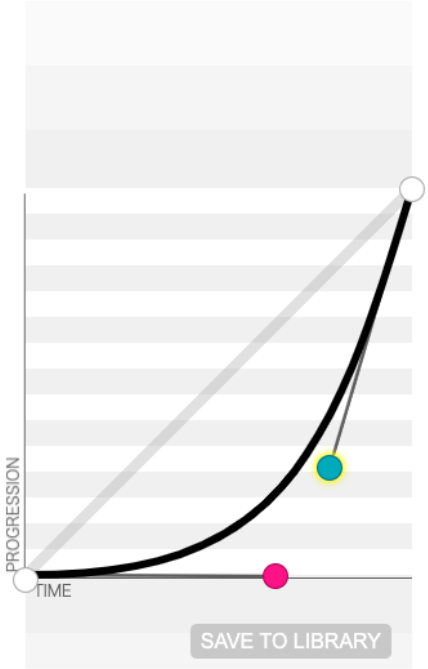






FOLLOW US

cubic-bezier(.65,0,.79,.28) COPY



SAVE TO LIBRARY

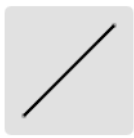
Preview & compare GO!

Duration: 0.1 seconds

Library IMPORT EXP

Click on a curve to compare it w

ease



linear

Tip: Right click on any library curve and
which you can share with others