CSS Keyframe Animations

Previously you learned some ways you can animate elements with CSS transitions and transforms, but what if you want things to happen in sequence? In order to do this, you'll need keyframe animation. That's what this handout will focus on: CSS keyframe animation.

CSS Keyframe Animation

In animation, a keyframe defines a starting or ending point of some action. For example, if a ball goes down then bounces up once and stops in the air, the keyframes would be the ball in the air (at the beginning), the ball at the bottom (at the moment of bounce), and again in the air (at the end). Keyframes are usually located at the extremes of motion (i.e. the "key" changes in the animation sequence). The in-between frames to get from point A to point B are just regular frames and not *key*frames.

In CSS, keyframe points are declared with percentages, so a keyframe at 50% is the keyframe at the halfway point of the sequence.

A basic CSS keyframe animation *rule* is declared as follows:

```
@keyframes ballmove {
  0% {
    top: 0;
  }
  50% {
    top: 100px;
    transform: scale(1);
  }
  100% {
    top: 0;
    transform: scale(2);
  }
}
```

In the above example, "ballmove" is the name of the animation sequence.

You've got an animation sequence defined but you'll need something to animate first. Create an HTML file with the following:

```
<!DOCTYPE html>
<html>
  <head>
    <title>CSS Keyframe Animation</title>
    <link rel="stylesheet" type="text/css" href="style.css" />
  </head>
  <body>
    <h1>CSS Keyframe Animations</h1>
    <h2>Ball 1</h2>
    <div class="animation-stage">
      <div class="ball" id="ball-1"></div>
    </div>
  </body>
</html>
```

Now in your CSS file, add the styling for the ball and animation-stage div in addition to the `@keyframes` rule. (For easier reading, I'll only be using the non-prefixed versions.)

```
html {
  box-sizing: border-box;
}
*, *:before, *:after {
  box-sizing: inherit;
}
.animation-stage {
  height: 150px;
  border:1px solid black;
}
.ball {
  border-radius: 50%;
  width: 50px;
  height: 50px;
}
#ball-1 {
  position: relative; /* so that you can use left, top, etc */
  background-color: red;
}
```

Now to define the set the animation on the ball itself, add the following bolded lines to #ball-1.

```
#ball-1 {
  position: relative;
  background-color: red;
  animation-name: ballmove;
  animation-duration: 5s;
  animation-iteration-count: 1; /* default */
  animation-direction: normal; /* default */
  animation-timing-function: ease; /* default */
  animation-fill-mode: none; /* default */
  animation-delay: 0s; /* default */
}
```

The animation properties above can be used in a shorthand version with the following format:

```
animation: [name] [duration] [iteration-count] [direction]
   [timing-function] [fill-mode] [delay];
```

The duration, timing-function, and delay properties work the same way as for transitions. The properties new to you are:

Property	**Description**
[name]	The name of your animation. This is the name you give your `@keyframes` rule.
[iteration-count]	This is how many times your animation should repeat. This can be a number (e.g. "2") or the "`infinite`" keyword for a forever-looping animation. The default is 1.
[direction]	There are four possible values: • **`normal`**: This is the default and just goes forward from 0% to 100%. If the animation loops, it will jump to 0% to begin the animation again. • **`reverse`**: The animation will go from 100% to 0%. • **`alternate`**: You will only see the effects of this if your animation will play more than once. It will start forward then backwards on the next iteration and so on. So on the first run, it will go from 0% to 100% and on the second run it will go from 100% to 0%. • **`alternate-reverse`**: This is similar to alternate, but it starts backwards then goes forward.
[fill-mode]	This defines whether styles will stay before or after an animation. There are four possible values: • **`none`**: This means that any styles in the animation will not apply to the element before or after the animation. This is the default. • **`backwards`**: This will apply when you use a delayed start. This makes the element take the relevant styles from the first keyframe of the animation even during the delay. Without setting this, the element will stay normal during the delay then jump values when the animation starts. • **`forwards`**: This will make the element keep the last styles from the animation even after the animation is ended. • **`both`**: This makes the fill-mode use both the forwards and backwards rules.

There are also vendor prefixes for each property so you can see why it is recommended to use the shorthand. You can change the `#ball-1` CSS block to use the animation shorthand. This will make the block:

```
#ball-1 {
  position: relative;
  background-color: red;
  animation: ballmove 5s;
}
```

Note that you don't need to specify properties if you are using the default values anyway.

<u>steps()</u>

It is possible to break your animation down into equal steps using the steps() function when you set your animation property. This is a type of timing function. For example, if you have an animation:

```
@keyframes dosomething {
  0% {
    top:0;
  }
  100% {
    top:50px;
  }
}
```

when you set your animation, you can automatically create 10 keyframes using the steps() function by:

```
animation: dosomething 5s steps(10);
```

This will automatically generate 10 keyframes, meaning each keyframe will move the element that's being animated down by 5px (i.e. (50px – 0px) / 10 = 5px) every 0.5 seconds (because 5s / 10 = 0.5s).

The `steps()` function is handy where you may want the movement to jump in steps. This function is what you would use to create a sprite sheet animation (see the link on Blackboard under **Diving Deeper**). To ensure the sprite animation is always running, just set the iteration count to infinite.

Multiple animations

You can make multiple animations apply to an element. Simply separate your animations with commas in the same animation statement.

```
animation: animation-1 2s, animation-2 3s;
```

Caution! Don't confuse this with combining transforms which only uses a space to separate transform <u>functions</u>.

animation-play-state

You can also specify the *state* of your animation via CSS. There are two possible state values:

- `running`
- `paused`

You can make your animation pause on hover by adding (using the first example):

```
.animation-stage:hover #ball-1 {
  animation-play-state: paused;
}
```

The above code means that when you hover over the stage, pause the ball animation. Notice that the selector must have `#ball-1` because the animation is on `#ball-1`.

There are vendor prefixes for this using -moz- and -webkit-. Remember when you use vendor prefixes, the last declaration should always be the non-prefixed version.

You can access this in Javascript with `[element].style.animationPlayState`. Vendor-prefixed versions will be like `webkitAnimationPlayState` instead of `animationPlayState`.

Using CSS3 animation with Javascript

When you are animating elements via CSS, certain events get fired:

- `animationstart`: Event fires when the animation begins.
- `animationiteration`: Event fires when at the beginning of each *iteration*. This applies when you have an animation which plays more than once.
- `animationend`: Event fires when the animation ends.

Note

The events above are the non-prefixed names of events. There are vendor-prefixed versions for older browsers for each event as shown below:

W3C Standard	Webkit	Mozilla	MS	Opera
animationstart	webkitAnimationStart	mozAnimationStart	MSAnimationStart	oAnimationStart
animationiteration	webkitAnimationIteration	mozAnimationIteration	MSAnimationIteration	oAnimationIteration
animationend	webkitAnimationEnd	mozAnimationEnd	MSAnimationEnd	oAnimationEnd

This means that when you add event listeners, you will need to add a listener for each prefixed and non-prefixed version.

The good news is that there is a prefixer helper function you can use so that you don't have add an event listener for every prefix. The following code was originally found in a post by Nick Salloum (page no longer available):

```
// prefixer helper function (code
var pfx = ["webkit", "moz", "MS", "o", ""];
function prefixedEventListener(element, type, callback) {
    for (var p = 0; p < pfx.length; p++) {
        if (!pfx[p]) type = type.toLowerCase();
        element.addEventListener(pfx[p]+type, callback, false);
    }
}
```

The above original code adds event listeners for all possible animation event names regardless of whether or not it's valid for the current browser. I have modified it so that only the valid event listener is added for the current browser (basing some of the logic off David Walsh's transitionend helper function):

```
function prefixedEventListener(element, type, callback) {
  //moved the "" first so that the standard version is checked first
  var pfx = ["", "webkit", "moz", "MS", "o"];
  var a = "Animation";
  for (var p = 0; p < pfx.length; p++) {
    if (!pfx[p]) {
      //animation CSS property
      a = a.toLowerCase();
      type = type.toLowerCase();
    }
    if (element.style[a] !== undefined) {
      element.addEventListener(pfx[p]+type, callback, false);
      return; //exit function on first valid event name
    }
  }
}
```

To use it, in your code, write:

```
prefixedEventListener([animated_element], "AnimationStart",
function(e) {
  //your function for animation start
  //this is your callback function
});
```

For the other events, just switch out "AnimationStart" for "AnimationIteration" or "AnimationEnd".

Being able to access CSS3 animation events via Javascript is useful because it allows you more fine-grained control. Using this, you can chain animations so that they happen one after the other rather than at the same time or you may want to animate stuff in combination with user interaction. For example, you want an animation to occur first *before* showing some content.

Using the first example of the ball we did, set the animation to only run three times. Now create an event listener to display a message saying that the animation is ended when it finishes running.
Create a Javascript file and make sure you add the file reference to your HTML page. Now in your Javascript file, add your event listener for `animationend`.

```
window.onload = function(e) {
  var ball = document.getElementById("ball-1");
  ball.addEventListener("animationend", displayMsg);
}
```

where displayMsg is a function which just displays an alert with the message "Animation ended".

```
function displayMsg() {
  alert("Animation ended");
}
```

Note that you only need the prefixer function in the grey "Note" box above if you need to support much older browsers.

Exercise

Make an animation of a bouncing ball and make it play infinitely. The ball should squish a little when it "bounces". Clicking on the stage should allow you to pause and play the animation. When paused, an alert should display how many times the ball bounce was animated.

Hint: You'll need an event listener for the `animationiteration` event.

Resources

https://thoughtbot.com/blog/css-animation-for-beginners

calc()

The **calc()** function is one which may come in handy when you are trying to move elements (especially when you are animating and the position from keyframe to keyframe is always changing). This function allows you do something like the following example.

For example, if you have an element with the following ID and dimensions:

```
#box {
  width:30px;
  height:30px;
}
```

and you are moving the element around inside a container, you can use in your animation (for #box):

```
top: calc(100% - 30px);
```

What the above property declaration will do is set the #box within its parent container so that its bottom edge will sit at the bottom edge of its parent container. If you set **`top:100%`** the box will be outside the parent container. Although in the above example, you could just use **`bottom:0`** this may not always be viable when animating.

The reason why is because you cannot mix `top` and `bottom` (and `left` and `right`) in your keyframes. For example, the following will not work:

```
@keyframes move {
  0% {
    top:0;
  }
  100% {
    bottom:0;
  }
}
```

In the above keyframes, running the animation won't do a thing. The element being animated will not move because you are trying to animate two different properties (top and bottom). From a computer's point of view, in the first keyframe `bottom` is undefined so how can the computer know where to move #box? Keep your animated properties consistent across keyframes (i.e. you can change the values to animate the changes, but the actual properties you're using should be consistent).

Challenge: I will show a Pacman keyframe animation challenge in class which you can try to figure out based on your current knowledge. (**Hint:** The calc() function will come in handy for this challenge.)