How to Use steps() in CSS Animations

Joni Trythall • CSS3, Tutorials • March 31, 2014 • 15 Comments

I am guessing that many of you have found`steps()` to be confusing when using it in CSS animations. I wasn' t sure how or why to use it at first and searching seems to produce two main examples: A typing demo by Lea Verou and an animated sprite sheet by Simurai.

These examples are genius and really helped me begin to understand this special little timing function, but they are such prominent references that it was hard to imagine how to use `steps()` outside of the context of each demo.

So, I really got into `steps()` and built a few animated demos to help those that might be as confused as I was in tackling this elusive beast.

Preview Download

Intro to Steps

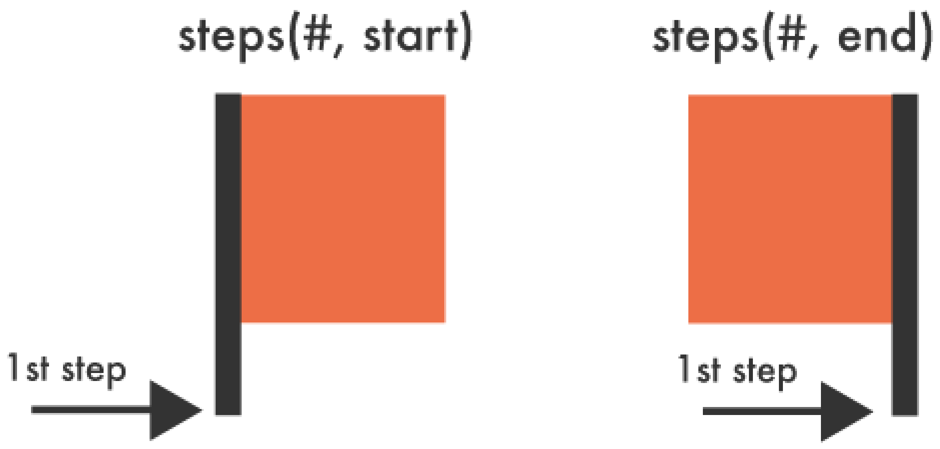
`steps()` is a timing function that allows us to break an animation or transition into segments, rather than one continuous transition from one state to another. The function takes two parameters — the first specifies the positive number of steps we want our animation to take.

```
steps(<number_of_steps>, <direction>)
```

The second parameter defines the point at which the action declared in our`@keyframes` will occur. This value is optional and will default to "end" if left unspecified. A direction of "start" denotes a left-continuous function and our animation' s first step will be completed as soon as the animation begins. It will jump immediately to the end of the first step and stay there until the end of this step duration.

A direction of "end" denotes a right-continuous function and directs the movement to stay put until the duration of the first step is completed. Each option essentially moves the element from a different side and will produce different positioning for the same animation.

Here' s a visual:



Impact of Fill Mode and Iteration Count

Before we get started it’ s important to understand how a different fill mode or iteration count will impact `steps()`, like the use of “forwards” or “infinite,” for example. If we have two cars with the same animation duration and the same `steps()` values, but one is directed to continue infinitely while the other the fills forward, the perceived end point of these two cars becomes very different, even though they start out from the same y-axis point.

The use of “forwards” commands the animated element to extend the styles from the last `@keyframes` of your animation to play beyond the duration of the animation and retain that state. Combining it with`steps()` in the animation makes the action appear as if the initial motionless state is not counting towards the total sum of steps, as it should with “end” , or that the car is taking an additional step beyond your`steps()` declarations, depending on how you look at it.

This sounds a bit messy but we will break it down in the demos. The main thing is to be mindful of how these variations will impact your intentions and number of steps. Here are some infinite versus forwards cars:

```
.contain-car {
   animation: drive 4s steps(
}

.contain-car-2 {
  animation: drive 4s steps(4
}
```

infinite
forwards

Now, let’ s take a look at some more code and try to make sense of this bit of trickery.

Steps Demos

You can take a look at the demos here, which consist of:

- A pure CSS clock
- Some energy efficient pure CSS cars
- Advancing bear paw prints
- A pure CSS progress circle

CSS Clock

A clock is ideal for demonstrating `steps()`. We need the clock’ s hands to rotate, but not in smooth and continuous movements. Using `steps()` will allow us to create a motion mimicking that of hands on a real clock.

There’ s a bit of math involved when using`steps()`, but it’ s not too painful. We want the second hand to to rotate 360 degrees through 60 steps and to complete the animation within 60 seconds.

```
.second {
  animation: tick-tock 60s st
}

@keyframes tick-tock {
  to {
    transform: rotate(360deg)
  }
}
```

The clock’ s animation declaration breaks down to 1 step per second.

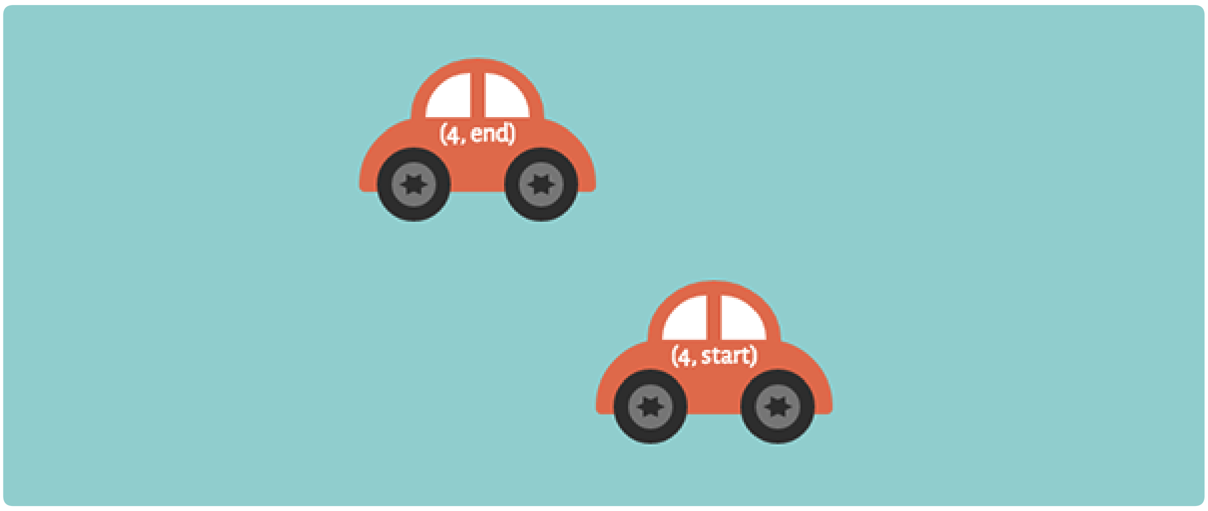
For the minute hand, we can assign the same `@keyframes`, but need different timing. We will multiply 60 by 60 to get our animation duration. Our hand will move once every 60 seconds, 60 times to complete a full 360-degree rotation.

```
.minute {
  animation: tick-tock 3600s 
}
```

And there it is!

Disclaimer: Don’ t rely on this clock to carry out your daily activities because, you know, it’ s a CSS clock.

CSS Cars



CSS cars demonstrate the difference between using “end” and “start” within our `steps()`. “Start” causes a car to move right away and stay until the duration of 1 step is complete. It will appear as if the “start” car is positioned further right than the “end” car, but if you add an animation delay to the second car you can see that both cars originate from the same y-axis point.

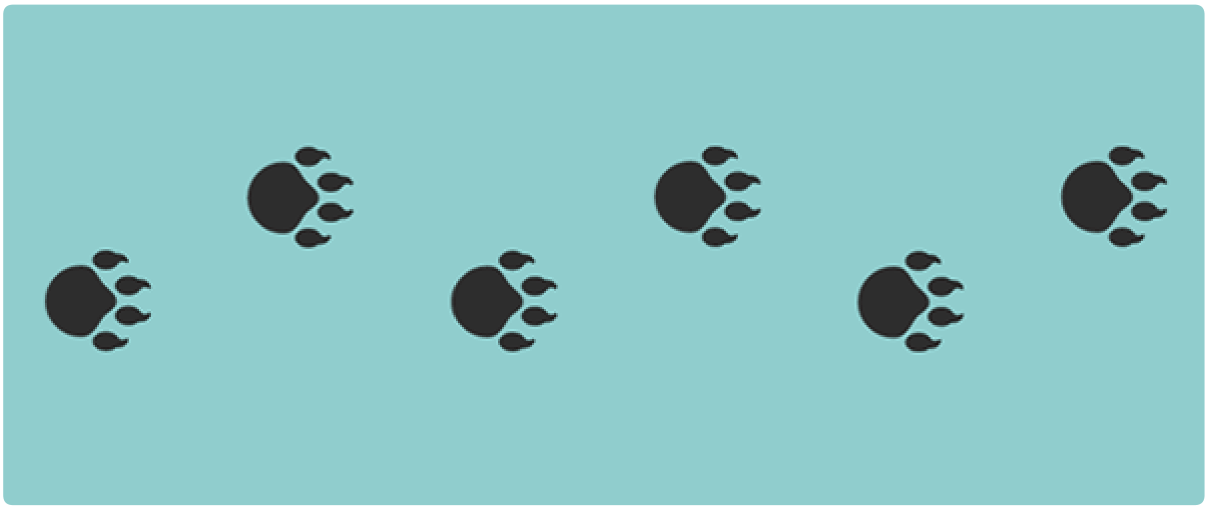
"End" causes the animation to essentially complete the designated time of 1 step before the animation begins its action. By the time our first car gets moving, it is on its second step so the cars have no chance of moving in sync. The white borders in the demo represent the start and end points of the animation.

```
.contain-car {
  animation: drive 4s steps(
}

.contain-car-2 {
  animation: drive 4s steps(
}

@keyframes drive {
  to {
    transform: translateX(64
  }
}
```

Bear Prints



Another way to better understand `steps()`is to take it quite literally and create actual steps. For this example we will be using bear paws. This demo uses one image consisting of six bear paw prints. This image is covered by a `<div>` and we want to move that `<div>` with `steps()` to reveal the paws in a way that mimics actual prints being left behind.

Without using `steps()` for this the `<div>`would move to the right in one fluid movement, which is not the effect we are going for. We want each paw to appear whole and at once.

As mentioned, there are six paw prints. We need to animate our `<div>` to move the full length of the image (675 px) while revealing one full print at a time.

```
.cover {
  animation: walk 7s steps(7,
}

@keyframes walk {
```