# This

The value of ‘this’ is evaluated during the run-time, depending on the context. The rule is simple: if ‘obj.f()’ is called, then ‘this’ is ‘obj’ during the call of ‘f’.

Arrow functions are special: they don’t have their “own” ‘this’. If we reference this from such a function, it’s taken from the outer “normal” function.

# This in JavaScript

There six different ways where ‘this’ can take on new values. They are:

1. ‘this’ in global context
2. ‘this’ in object construction
3. ‘this’ in an object method
4. ‘this’ in a simple function
5. ‘this’ in an arrow function
6. ‘this’ in an event listener

When ‘this’ is called outside of any function, in a global context, ‘this’ defaults to the ‘Window’ object in the browser. Usually, you wouldn’t use ‘this’ in a global context anyway.

## ‘this’ in object construction

When you create a new instance of an object with the ‘new’ keyword, ‘this’ refers to the instance.

Example: function Person(*age*) {

*this*.age = age

}

const greg = **new** Person(22)

console.log(greg) // in this instance “this” refers to “greg” therefore this.age = greg.age = 22.

## ‘this’ in an object method

‘this’ within any method refers to the object itself.

Example: let o = {

sayThis() {

console.log(this) // return as o

},

}

## ‘this’ in a simple function

On browsers, ‘this’ is always set to ‘Window’ in a simple function. One quick fix is to create a variable that stores the reference to the ‘this’. This variable is often called ‘self’ or ‘that’.

## ‘this’ in an arrow function

‘this’ in an arrow function is always the same as ‘this’ around it (in its immediate scope). So, if you use arrow functions within an object method, the ‘this’ context stays as the object, not Window.

With arrow functions, the speakLeet example could be written in the following way:

const o = {

doSomethingLater() {

setTimeout(() => this.speakLeet(), 1000)

},

speakLeet() {

console.log(`1337 15 4W350M3`)

},

}

A third way to change the value of ‘this’ within any function is to use either ‘bind’, ‘call’ or ‘apply’.

## ‘this’ in an event listener

‘this’ is set to the element that fired the event in an event listener:

Example: let button = document.querySelector('button')

button.addEventListener('click', function () {console.log(this) // button})

## Bind

‘bind’ is a method that is present in every function. It allows you to change the ‘this’ context. This method takes in any number of arguments and returns the bound function. The first parameter you pass into ‘bind’ becomes ‘this’ in the bound function. Once you have created a bound function, you can call it anytime you wish. The other parameters you pass to ‘bind’ will be passed as arguments to the original function.

‘bind’ doesn’t work with arrow functions.

# CSS-animations

The idea of CSS transitions is simple. We describe a property and how its changes should be animated. When the property changes, the browser paints the animation.

There are 4 properties to describe CSS transitions:

1. transition-property
2. transition-duration
3. transition-timing-function
4. transition-delay

We’ll cover them in a moment, for now let’s note that the common transition property allows declaring them together in the order: property duration timing-function delay, as well as animating multiple properties at once.

In transition-property, we write a list of properties to animate, for instance: left, margin-left, height, color. Or we could write all, which means “animate all properties”.

In transition-duration we can specify how long the animation should take. The time should be in CSS time format: in seconds (s) or milliseconds (ms).

In transition-delay we can specify the delay *before* the animation. For instance, if transition-delay is 1s and transition-duration is 2s, then the animation starts 1 second after the property change and the total duration will be 2 seconds. Negative values are also possible. Then the animation is shown immediately, but the starting point of the animation will be after given value (time). For example, if transition-delay is -1s and transition-duration is 2s, then animation starts from the halfway point and total duration will be 1 second.

The transform property is animated like this:

#stripe.animate {

transform: translate(-90%);

transition-property: transform;

transition-duration: 9s;

}

In the example above JavaScript adds the class .animate to the element – and the animation starts:

stripe.classList.add('animate');

We could also start it from somewhere in the middle of the transition, from an exact number, e.g. corresponding to the current second, using a negative transition-delay.

The timing function describes how the animation process is distributed along its timeline. Will it start slowly and then go fast, or vice versa. It appears to be the most complicated property at first. But it becomes very simple if we devote a bit time to it. That property accepts two kinds of values: a Bezier curve or steps.

The timing function can be set as a Bezier curve with 4 control points that satisfy the conditions:

1. First control point: (0,0).
2. Last control point: (1,1).
3. For intermediate points, the values of x must be in the interval 0..1, y can be anything.

The syntax for a Bezier curve in CSS: cubic-bezier(x2, y2, x3, y3). Here we need to specify only 2nd and 3rd control points, because the 1st one is fixed to (0,0) and the 4th one is (1,1).

The timing function describes how fast the animation process goes. The x axis is the time: 0 – the start, 1 – the end of transition-duration. The y axis specifies the completion of the process: 0 – the starting value of the property, 1 – the final value. The simplest variant is when the animation goes uniformly, with the same linear speed. That can be specified by the curve cubic-bezier(0, 0, 1, 1).

There are several built-in curves: linear, ease, ease-in, ease-out and ease-in-out. The linear is a shorthand for cubic-bezier(0, 0, 1, 1) – a straight line.

A screenshot of a computer

Description automatically generated with medium confidence

The timing function steps(number of steps[, start/end]) allows splitting an transition into multiple steps. In the HTML, a stripe of digits is enclosed into a fixed-length <div id="digits">.

The first argument of steps(9, start) is the number of steps. The transform will be split into 9 parts (10% each). The time interval is automatically divided into 9 parts as well, so transition: 9s gives us 9 seconds for the whole animation – 1 second per digit.

The second argument is one of two words: start or end.

The start means that in the beginning of animation we need to make the first step immediately.

When the CSS animation finishes, the transitionend event triggers. It is widely used to do an action after the animation is done. Also we can join animations.

The event object for transitionend has a few specific properties:

1. **event.propertyName -** The property that has finished animating. Can be good if we animate multiple properties simultaneously.
2. **event.elapsedTime -** The time (in seconds) that the animation took, without transition-delay.

We can join multiple simple animations together using the @keyframes CSS rule. It specifies the “name” of the animation and rules – what, when and where to animate. Then using the animation property, we can attach the animation to the element and specify additional parameters for it.

A screenshot of a computer program

Description automatically generated with medium confidence

In more technical details, when there’s a style change, the browser goes through 3 steps to render the new look:

1. Layout: re-compute the geometry and position of each element, then
2. Paint: re-compute how everything should look like at their places, including background, colors,
3. Composite: render the final results into pixels on screen, apply CSS transforms if they exist.

During a CSS animation, this process repeats every frame. However, CSS properties that never affect geometry or position, such as color, may skip the Layout step. If a color changes, the browser doesn’t calculate any new geometry, it goes to Paint → Composite. And there are few properties that directly go to Composite. You can find a longer list of CSS properties and which stages they trigger at [https://csstriggers.com](https://csstriggers.com/).

Animations of properties that skip the Layout step are faster. It’s even better if Paint is skipped too. The transform property is a great choice, because:

* CSS transforms affect the target element box as a whole (rotate, flip, stretch, shift it).
* CSS transforms never affect neighbour elements.

…So browsers apply transform “on top” of existing Layout and Paint calculations, in the Composite stage. In other words, the browser calculates the Layout (sizes, positions), paints it with colors, backgrounds, etc at the Paint stage, and then applies transform to element boxes that need it. Changes (animations) of the transform property never trigger Layout and Paint steps. More than that, the browser leverages the graphics accelerator (a special chip on the CPU or graphics card) for CSS transforms, thus making them very efficient. Luckily, the transform property is very powerful. By using transform on an element, you could rotate and flip it, stretch and shrink it, move it around, and much more. So instead of left/margin-left properties we can use transform: translateX(…), use transform: scale for increasing element size, etc.

The opacity property also never triggers Layout (also skips Paint in Mozilla Gecko). We can use it for show/hide or fade-in/fade-out effects.

Paring transform with opacity can usually solve most of our needs, providing fluid, good-looking animations.