With regular functions, the value of this is set based on *how the function is called*. With arrow functions, the value of this is based on *the function's surrounding context*. In other words, the value of this *inside* an arrow function is the same as the value of this *outside* the function.

Let's check out an example with this in regular functions and then look at how arrow functions will work.

*// constructor*

**function** **IceCream**() {

**this**.scoops = 0;

}

*// adds scoop to ice cream*

IceCream.prototype.addScoop = **function**() {

setTimeout(**function**() {

**this**.scoops++;

console.log('scoop added!');

}, 500);

};

**const** dessert = **new** IceCream();

dessert.addScoop();

***Prints:*** *scoop added!*

After running the code above, you'd *think* that dessert.scoops would be 1 after half a millisecond. But, unfortunately, it's not:

console.log(dessert.scoops);

***Prints:*** *0*

Can you tell why?

The function passed to setTimeout() is called without new, without call(), without apply(), and without a context object. That means the value of this inside the function is the global object and **NOT** the dessert object. So what actually happened was that a new scoops variable was created (with a default value of undefined) and was then incremented (undefined + 1 results in NaN):

console.log(scoops);

***Prints:*** *NaN*

One way around this is to use closure:

*// constructor*

**function** **IceCream**() {

**this**.scoops = 0;

}

*// adds scoop to ice cream*

IceCream.prototype.addScoop = **function**() {

**const** cone = **this**; *// sets `this` to the `cone` variable*

setTimeout(**function**() {

cone.scoops++; *// references the `cone` variable*

console.log('scoop added!');

}, 0.5);

};

**const** dessert = **new** IceCream();

dessert.addScoop();

The code above *will* work because instead of using this inside the function, it sets the cone variable to this and then looks up the cone variable when the function is called. This works because it's using the value of the this outside the function. So if we check the number of scoops in our dessert right now, we'll see the correct value of 1:

console.log(dessert.scoops);

***Prints:*** *1*

Well that's exactly what arrow functions do, so let's replace the function passed to setTimeout() with an arrow function:

*// constructor*

**function** **IceCream**() {

**this**.scoops = 0;

}

*// adds scoop to ice cream*

IceCream.prototype.addScoop = **function**() {

setTimeout(() => { *// an arrow function is passed to setTimeout*

**this**.scoops++;

console.log('scoop added!');

}, 0.5);

};

**const** dessert = **new** IceCream();

dessert.addScoop();

Since arrow functions inherit their this value from the surrounding context, this code works!

console.log(dessert.scoops);

***Prints:*** *1*

When addScoop() is called, the value of this *inside* addScoop() refers to dessert. Since an arrow function is passed to setTimeout(), it's using its surrounding context to determine what this refers to inside itself. So since this *outside* of the arrow function refers to dessert, the value of this *inside* the arrow function will also refer to dessert.

Now what do you think would happen if we changed the addScoop() method to an arrow function?

*// constructor*

**function** **IceCream**() {

**this**.scoops = 0;

}

*// adds scoop to ice cream*

IceCream.prototype.addScoop = () => { *// addScoop is now an arrow function*

setTimeout(() => {

**this**.scoops++;

console.log('scoop added!');

}, 0.5);

};

**const** dessert = **new** IceCream();

dessert.addScoop();

Yeah, this doesn't work for the same reason - arrow functions inherit their this value from their surrounding context. Outside of the addScoop() method, the value of this is the global object. So if addScoop() is an arrow function, the value of this *inside* addScoop() is the global object. Which then makes the value of this in the function passed to setTimeout() *also* set to the global object!