What is “this” keyword in JavaScript

Let’s go back to the school days when we learned about pronouns.

```
Phelps is swimming fast because he wants to win the race.
```

Note the use of the pronoun “he”. We don’t directly address Phelps here but use the pronoun he to refer to Phelps. Similarly, JavaScript uses the `this` keyword as a reference to the object in context i.e the subject.

Example:

```
var car= {
make: "Lamborghini",
model: "Huracán",
fullName: function () {
console.log(this.make+" " +this.model);
console.log(car.make+ " " +car.model);
}
}
car.fullName();

//Lamborghini Huracán
//Lamborghini Huracán
```

In the above code, we have an object car that has the properties make, model and fullName. The value of fullName is a function that prints the full name of the car using 2 different syntaxes.

- Using this as in `this.make+” “ +this.model` , the *this* refers to the object in context (which is car) so `this.make` is effectively `car.make` and so is `this.model` .
- Using dot(.) notation, we can also access the properties of objects, `car.make` & `car.model` .

Hence, both the syntax results in the same result in the console. To understand `this` binding, we have to understand the **call-site**: the *location* in code where a function is **called** (not where it's declared). This is because, how `this` binds depends upon the context or the call-site. To illustrate what is call-site, look at the following code:

```
function baz() {
    console.log( "baz" );
    bar(); // <-- call-site for `bar`
}

function bar() {
    console.log( "bar" );
    foo(); // <-- call-site for `foo`
}

function foo() {
    console.log( "foo" );
}

baz(); // <-- call-site for `baz`
```

Let us understand more about `this` keyword and how it binds depending upon the call-site through the following example:

```
function foo() {
    var a = 2;
    this.bar();
}

function bar() {
    console.log( this.a );
}

foo(); //undefined
```

When the function `foo` is invoked, it calls the function `this.bar` implicitly. Since the calling site of `foo` is *global scope*, the Engine binds `this.bar` to the global

scope and finds the function `bar` in the global scope which is found. Now, the function `bar` itself calls `console.log(this.a)` and calling site of the function is `global` and the Engine does not find the variable `a` in the global scope, thus when the function `foo` is invoked, it returns `undefined`.

Another example

`this` can also be binded using an object reference. Let us take an example to understand.

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
foo();     // undefined
```

The call-site of `foo()` is although *global* but it is referenced by an object `obj` which binds `this` to `obj` and thus could find the variable `a` in its scope and prints `a`.

But when `foo()` is called without any reference, it printed `undefined` as it binds `this` to the global scope(call-site of `foo`) and found no variable named `a`.

Consider another example to elaborate:

```
function foo(num) {
    console.log( "foo: " + num );
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// how many times was `foo` called?
console.log( foo.count ); // 0
```

Why did `console.log(foo.count)` gave the result `0`, although `foo` is called `4` times? This is because `foo` is called in *global scope* and thus `this.count` in the function `foo` is bind to the call site of `foo` which is `global` and thus `this.count` in the function `foo` creates a global variable `count` whose value is incremented to `4`. Moreover, `foo.count` is another variable that is bound to function `foo` and it has nothing to do with `this.count`.