**The ‘this’ keyword: A Gentle Explanation in JavaScript -** [Jan Cibulka](https://medium.com/@jannden?source=post_page-----bdb30a68a5d2--------------------------------) Mar 1, 2022

The **this**keyword can be quite confusing in JavaScript.

In object-oriented programming languages, it usually refers to the current class instance. But JavaScript has a completely different approach. The value of **this**keyword depends on how a function, which uses it, is called.

I will go through a set of examples to demonstrate how **this** works. You can find a full concise code in my [GitHub Gist](https://gist.github.com/jannden/508fe904423cb9b9653d671cf93e98bb).

Imagine a simple JavaScript object.

const sorceress = {  
 givenName: "Yennefer",  
};

We might want to bring it alive by adding a property that does something.

// ...  
sorceress.does = function(someSpell) {  
 console.log(`Yennefer casts ${ someSpell }.`)  
}  
sorceress.does("fireball");  
// → Yennefer casts fireball.

As you can see, the name of the sorceress — Yennefer — is hardcoded in the text. Isn’t it a shame, in case the object sorceress has a property givenName which holds the value Yennefer? But how to access that text property givenName from the functional property does()?

We can use the whole path, such as:

// ...  
sorceress.does = function(someSpell) {  
 console.log(`${ sorceress.givenName } casts ${ someSpell }.`)  
}

But we have a better option. Since both properties (givenName and does()) are of the same object, we can use the **this**keyword. We will get the same result in a much cleaner way:

// ...  
sorceress.does = function(someSpell) {  
 console.log(`${ **this**.givenName } casts ${ someSpell }.`)  
}

Let’s have a look at the whole example to make sure we grasp the concept. We will create a new object with the text property givenName and a functional property does() that refers to its parent object with the **this**keyword:

const wizard = {  
 givenName: "Mousesack",  
 does: function(someSpell) {  
 console.log(`${ this.givenName } casts ${ someSpell }.`);  
 },  
};  
wizard.does("protective shield");  
// → Mousesack casts protective shield.

The functional property doesn’t even have to be defined from within the object. We can define it outside of it as a normal function. The keyword **this**will take different forms based on where the function is called from.

function ruledArea(country) {  
 console.log(`${ this.givenName } rules ${ country }.`);  
}const queen = {  
 givenName: "Calanthe",  
 rules: ruledArea  
}  
queen.rules("Cintra");  
// Here, this.givenName points to "Calanthe"  
// → Calanthe rules Cintra.const king = {  
 givenName: "Foltest",  
 rules: ruledArea  
}  
king.rules("Temeria");  
// Here, this.givenName points to "Foltest"  
// → Foltest rules Temeria.

Consider now the function ruledArea. Obviously, it receives a hidden parameter that points to its parent object. If we don’t call it from any parent object, this points to the browser window itself.

**The Call method**

ruledArea is a function and so has its own **prototype** filled with useful properties from the Function.prototype object (if you don’t understand why, read my [gentle explanation of prototypes](https://medium.com/@jannden/prototypes-a-gentle-explanation-in-javascript-470eb0e372af)). One of such properties is the method named call() which we can use to set a parent object of our function and immediately call it:

// ...  
function secondRuledArea(country) {  
 console.log(`${ this.givenName } also rules ${ country }.`);  
}  
secondRuledArea.**call**(king, "Pontaria");  
// → Foltest also rules Pontaria.

As we see, each function has its own **this**binding. So the scope doesn’t determine, what the keyword **this**means. For example, here is an example with incorrect:

// ...  
function rulesOver(...cities) {  
 cities.map(**function anotherFunction(city)** {  
 console.log(  
 `${ city } is ruled by ${ this.givenName }`  
 );  
 })  
}  
rulesOver.**call**(king, "Vizima", "Brugge");  
// → Vizima is ruled by **undefined**  
// → Brugge is ruled by **undefined**

We see that **this**didn’t point to the object king. It actually pointed to the browser window, as the function anotherFunction() wasn’t called as a property of another object.

I have said that each function has its own **this**binding, even if it points just to the browser window. However, there is an exception — the arrow functions. They don’t have their own **this**binding, so we can use the upper scope’s binding. Let’s rewrite the previous anotherFunction() function as an anonymous arrow function:

// ...  
function rulesOver(...cities) {  
 cities.map((city) =>  
 console.log(`The city ${city} is ruled by ${this.givenName}`)  
 );  
}  
rulesOver.**call**(king, "Vizima", "Brugge");  
// → Vizima is ruled by Foltest  
// → Brugge is ruled by Foltest

**The Apply method**

The apply() method is similar to call(). The difference is that the apply() method accepts an array of arguments instead of comma-separated values.

// …  
rulesOver.**apply**(king, [“Vizima”, “Brugge”]); **// passing an array**  
// → Vizima is ruled by Foltest  
// → Brugge is ruled by Foltest

**The Bind Method**

Another useful method is bind(). It sets the **this**keyword to whatever we choose, but it doesn’t immediately call the function itself.

When we pass the functional properties around, **this** gets unbound from the parent object. For example, this is just a slightly modified version of the wizard object from earlier, but now it won’t work due to passing wizard.does as an unbound function to a new variable:

const wizard = {  
 givenName: “Mousesack”,  
 does: function(someSpell) {  
 console.log(`${ this.givenName } casts ${ someSpell }.`);  
 },  
};  
**const unboundDoes = wizard.does;**  
**unboundDoes(“protective shield”);**  
// → **undefined** casts protective shield.

We can fix it with the bind() method easily:

// …  
const boundDoes = **unboundDoes.bind(wizard)**;  
boundDoes(“protective shield”);  
// → Mousesack casts protective shield.

**Conclusion**

We have learned that the way the **this**keyword behaves is relative to how a function, which uses it, is called. We can override the behavior of **this**with the help of the call, apply, and bind methods.