**Conceptual Overview of "this"**

When a function is created, a keyword called this is created (behind the scenes), which links to the object in which the function operates.

**The this keyword’s value has nothing to do with the function itself, how the function is called determines this's value**

**Default "this" context**

*// define a function*

**var** myFunction = **function** () {

console.log(**this**);

};

*// call it*

myFunction();

What can we expect the this value to be? By default, this should always be the window Object, which refers to the root—the global scope, except **if the script is running in strict mode ("use strict") this will be undefined.**

## Object literals

**var** myObject = {

myMethod: **function** () {

console.log(**this**);

}

};

What would be the this context here?...

-- this === myObject?

-- this === window?

-- this === anything else?

Well, the answer is **We do not know!** Remember:**The this keyword’s value has nothing to do with the function itself, how the function is called determines the this value**

Okay, let's change the code a bit...

**var** myMethod = **function** () {

console.log(**this**);

};

**var** myObject = {

myMethod: myMethod

};

Is it clearer now? Of course, everything depends on how we call the function.

myObject in the code is given a property called myMethod, which points to the myMethod function. When the myMethod function is called from the global scope, this refers to the window object. When it is called as a method of myObject, this refers to myObject.

myObject.myMethod() *// this === myObject*

myMethod() *// this === window*

This is called **implicit binding**

**Explicit binding( call() / apply() )** is when we explicitly bind a context to the function. This is done with call() or apply()

**var** myMethod = **function** () {

console.log(**this**);

};

**var** myObject = {

myMethod: myMethod

};

myMethod() *// this === window*

myMethod.call(myObject, args1, args2, ...) *// this === myObject*

myMethod.apply(myObject, [array **of** args]) *// this === myObject*

***Which is more precedent, implicit binding or explicit binding??????????????***

**var** myMethod = **function** ()

**console**.**log**(this);

};

**var** **obj1** = {

a: 2,

myMethod: myMethod

};

**var** obj2 = {

a: 3,

myMethod: myMethod

};

obj1.myMethod(); *// 2*

obj2.myMethod(); *// 3*

obj1.myMethod.call( obj2 ); *// ?????*

obj2.myMethod.call( obj1 ); *// ?????*

***Explicit binding takes precedence over implicit binding, which means you should ask first if explicit binding applies before checking for implicit binding.***

obj1.myMethod.call( obj2 ); *// 3*

obj2.myMethod.call( obj1 ); *// 2*

### Hard binding

This is done with bind() (ES5). bind() returns a new function that is hard-coded to call the original function with the this context set as you specified.

myMethod = myMethod.bind(myObject);

myMethod(); *// this === myObject*

***Hard binding takes precedence over explicit binding.***

**var** myMethod = **function** () {

console.log(**this**);

};

**var** obj1 = {

a: 2

};

**var** obj2 = {

a: 3

};

myMethod = myMethod.bind(obj1); *// 2*

myMethod.call( obj2 ); *// 2*

### ‘New’ binding

**function** **foo**(a) {

**this**.a = a;

}

**var** bar = **new** foo( 2 );

console.log( bar.a ); *// 2*

So this when the function has been called with new refers to the new instance created.

When a function is called with **new**, it does not care about implicit, explicit, or hard binding, it just creates the new context—which is the new instance.

**function** **foo**(something) {

**this**.a = something;

}

**var** obj1 = {};

**var** bar = foo.bind( obj1 );

bar( 2 );

console.log( obj1.a ); *// 2*

**var** baz = **new** bar( 3 );

console.log( obj1.a ); *// 2*

console.log( baz.a ); *// 3*

### API calls

Sometimes, we use a library or a helper object which does something (Ajax, event handling, etc.) and it calls a passed callback. Well, we have to be careful in these cases. Example:

myObject = {

myMethod: **function** () {

helperObject.doSomethingCool('superCool',

**this**.onSomethingCoolDone);

},

onSomethingCoolDone: **function** () {

*/// Only god knows what is "this" here*

}

};

Take a look at the code. You might think that because we are passing "this.onSomethingCoolDone" as a callback, the scope is passing a reference to that method and not to the way to call it.

To fix this, there are a few ways:

1. **Usually libraries offer another parameter, so then you can pass the scope you want to get back.**

myObject = {

myMethod: **function** () {

helperObject.doSomethingCool('superCool', **this**.onSomethingCoolDone, **this**);

},

onSomethingCoolDone: **function** () {

*/// Now everybody know that "this" === myObject*

}

};

1. **You can hard bind the method to the scope you want (ES5)**

myObject = {

myMethod: **function** () {

helperObject.doSomethingCool('superCool', **this**.onSomethingCoolDone.bind(**this**));

},

onSomethingCoolDone: **function** () {

*/// Now everybody know that "this" === myObject*

}

};

1. **You can create a closure and cache this into me. For example:**

myObject = {

myMethod: **function** () {

**var** me = **this**;

helperObject.doSomethingCool('superCool', **function** () {

*/// Only god knows what is "this" here, but we have access to "me"*

});

}

};

I do not recommend this approach because it can cause memory leaks and it tends to make you forget about the real scope and rely on variables. You can get to the point where your scope is a real mess.

This problem applies also to event listeners, timeouts, forEach, etc.

Summary:

1. The this keyword’s value has nothing to do with the function itself, how the function is called determines the this value. this is not assigned a value until an object invokes the function where this is defined
2. It can be dynamic, based on how the function is called
3. You can change the this context through .call(), .apply() and .bind()

## Implicit Binding

this points to the object on which the function is called (what’s to the left of the period when the function is called).

## Explicit Binding

We can explicitly tell the JavaScript engine to set this to point to a certain value using .call(), .apply() or .bind().