How does "this" keyword work within a function?

Asked 16 years, 3 months ago Modified 2 years ago Viewed 106k times

design decision/philosophy)? Pared-down code example:



257

I just came across an interesting situation in JavaScript. I have a class with a method that defines several objects using object-literal notation. Inside those objects, the this pointer is being used. From the behavior of the program, I have deduced that the this pointer is referring to the class on which the method was invoked, and not



the object being created by the literal.

This seems arbitrary, though it is the way I would expect it to work. Is this defined behavior? Is it cross-browser safe? Is there any reasoning underlying why it is the way it is beyond "the spec says so" (for instance, is it a consequence of some broader

```
// inside class definition, itself an object literal, we have this function:
onRender: function() {
    this.menuItems = this.menuItems.concat([
            text: 'Group by Module',
            rptletdiv: this
        },
            text: 'Group by Status',
            rptletdiv: this
        }]);
    // etc
}
```

iavascript this language-design language-features

Share Improve this question Follow

```
edited Jun 21, 2022 at 13:35
      Willem van der Veen
      36.4k • 18 • 204 • 176
```

```
asked Sep 25, 2008 at 15:25
  rmeador
    25.6k • 18 • 64 • 104
```

```
it does happen when i do this too var signup = {
onLoadHandler:function(){
                                                console.log(this);
return Type.createDelegate(this,this._onLoad);
_onLoad: function (s, a) {
                                              console.log("this",this);
}}; - Deeptechtons Oct 18, 2011 at 10:43 <a>▶</a>
see also How does the "this" keyword in Javascript act within an object literal? - Bergi Jul 30,
2015 at 14:13
```

check out this post. Has some good explanation of various usage and behaviors of this keyword. – Love Hasija Aug 31, 2017 at 2:31 ✓

checkout this link scotch.io/@alZami/understanding-this-in-javascript - AL-zami Sep 11, 2017 at 5:21

7 Answers

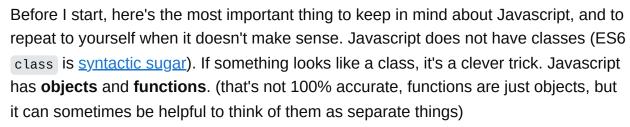
Sorted by: Highest score (default)



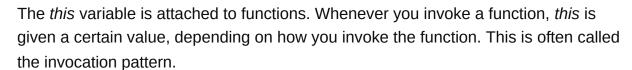
Cannibalized from another post of mine, here's more than you ever wanted to know about *this*.

577









There are four ways to invoke functions in javascript. You can invoke the function as a *method*, as a *function*, as a *constructor*, and with *apply*.

As a Method

A method is a function that's attached to an object

```
var foo = {};
foo.someMethod = function(){
    alert(this);
}
```

When invoked as a method, *this* will be bound to the object the function/method is a part of. In this example, this will be bound to foo.

As A Function

If you have a stand alone function, the *this* variable will be bound to the "global" object, almost always the *window* object in the context of a browser.

```
var foo = function(){
   alert(this);
}
```

```
foo();
```

This may be what's tripping you up, but don't feel bad. Many people consider this a bad design decision. Since a callback is invoked as a function and not as a method, that's why you're seeing what appears to be inconsistent behavior.

Many people get around the problem by doing something like, um, this

```
var foo = {};
foo.someMethod = function (){
   var that=this;
   function bar(){
      alert(that);
   }
}
```

You define a variable *that* which points to *this*. Closure (a topic all its own) keeps *that* around, so if you call bar as a callback, it still has a reference.

NOTE: In use strict mode if used as function, this is not bound to global. (It is undefined).

As a Constructor

You can also invoke a function as a constructor. Based on the naming convention you're using (TestObject) this also **may be what you're doing and is what's tripping you up**.

You invoke a function as a Constructor with the new keyword.

```
function Foo(){
    this.confusing = 'hell yeah';
}
var myObject = new Foo();
```

When invoked as a constructor, a new Object will be created, and *this* will be bound to that object. Again, if you have inner functions and they're used as callbacks, you'll be invoking them as functions, and *this* will be bound to the global object. Use that var that = this trick/pattern.

Some people think the constructor/new keyword was a bone thrown to Java/traditional OOP programmers as a way to create something similar to classes.

With the Apply Method

Finally, every function has a method (yes, functions are objects in Javascript) named "apply". Apply lets you determine what the value of *this* will be, and also lets you pass in an array of arguments. Here's a useless example.

```
function foo(a,b){
    alert(a);
    alert(b);
    alert(this);
}
var args = ['ah','be'];
foo.apply('omg',args);
```

Share

Improve this answer

Follow



answered Sep 25, 2008 at 15:52

Alana Storm



- Note: In <u>strict mode</u>, this will be <u>undefined</u> for function invocations. Miscreant Jul 8, 2015 at 18:05
- 1 A function declaration, eg. function myfunction () {}, is a special case of "as a method" where "this" is the global scope (window). − richard Jul 23, 2015 at 6:54 ✓
- 1 @richard: Except in strict mode, and this has nothing to do with scope. You mean the global *object*. T.J. Crowder Jul 18, 2016 at 13:45

@alan-storm . In the case of "As a constructor" is this.confusing = 'hell yeah'; the same as var confusing = 'hell yeah'; ? So both will allow my0bject.confusing ? It would be nice if not just so that you can use this to create the properties and other variables for the internal work. — wunth Sep 20, 2017 at 4:23

But then again I guess that the working stuff can be done outside the function and the value passed through to the constructor: function Foo(thought){ this.confusing = thought; } and then var myObject = new Foo("hell yeah"); - wunth Sep 20, 2017 at 19:21



Function calls

Functions are just a type of Object.



36

All Function objects have <u>call</u> and <u>apply</u> methods which execute the Function object they're called on.



When called, the first argument to these methods specifies the object which will be referenced by the this keyword during execution of the Function - if it's null or undefined, the global object, window, is used for this.

Thus, calling a Function...

```
whereAmI = "window";

function foo()
{
    return "this is " + this.whereAmI + " with " + arguments.length + " +
    arguments";
}
```

...with parentheses - foo() - is equivalent to foo.call(undefined) or foo.apply(undefined), which is *effectively* the same as foo.call(window) or foo.apply(window).

```
>>> foo()
"this is window with 0 arguments"
>>> foo.call()
"this is window with 0 arguments"
```

Additional arguments to call are passed as the arguments to the function call, whereas a single additional argument to apply can specify the arguments for the function call as an Array-like object.

```
Thus, foo(1, 2, 3) is equivalent to foo.call(null, 1, 2, 3) or foo.apply(null, [1, 2, 3]).
```

```
>>> foo(1, 2, 3)
"this is window with 3 arguments"
>>> foo.apply(null, [1, 2, 3])
"this is window with 3 arguments"
```

If a function is a property of an object...

```
var obj =
{
    whereAmI: "obj",
    foo: foo
};
```

...accessing a reference to the Function via the object and calling it with parentheses - obj.foo() - is equivalent to foo.call(obj) or foo.apply(obj).

However, functions held as properties of objects are not "bound" to those objects. As you can see in the definition of obj above, since Functions are just a type of Object, they can be referenced (and thus can be passed by reference to a Function call or returned by reference from a Function call). When a reference to a Function is passed, no additional information about where it was passed *from* is carried with it, which is why the following happens:

```
>>> baz = obj.foo;
>>> baz();
"this is window with 0 arguments"
```

The call to our Function reference, baz, doesn't provide any context for the call, so it's effectively the same as baz.call(undefined), so this ends up referencing window. If we want baz to know that it belongs to obj, we need to somehow provide that information when baz is called, which is where the first argument to call or apply and closures come into play.

Scope chains

```
function bind(func, context)
{
    return function()
    {
        func.apply(context, arguments);
    };
}
```

When a Function is executed, it creates a new scope and has a reference to any enclosing scope. When the anonymous function is created in the above example, it has a reference to the scope it was created in, which is bind's scope. This is known as a "closure."

When you attempt to access a variable this "scope chain" is walked to find a variable with the given name - if the current scope doesn't contain the variable, you look at the next scope in the chain, and so on until you reach the global scope. When the anonymous function is returned and bind finishes executing, the anonymous function still has a reference to bind 's scope, so bind 's scope doesn't "go away".

Given all the above you should now be able to understand how scope works in the following example, and why the technique for passing a function around "pre-bound" with a particular value of this it will have when it is called works:

```
>>> baz = bind(obj.foo, obj);
>>> baz(1, 2);
"this is obj with 2 arguments"
```



"When a reference to a Function is passed, no additional information about where it was passed from is carried with it" thank you @insin for this. – Alex Marandon Mar 18, 2012 at 12:03



Is this defined behavior? Is it cross-browser safe?



Yes. And yes.



Is there any reasoning underlying why it is the way it is...



The meaning of this is pretty simple to deduce:

- 1. If this is used inside a constructor function, and the function was invoked with the new keyword, this refers to the object that will be created. this will continue to mean the object even in public methods.
- 2. If this is used anywhere else, including nested *protected* functions, it refers to the global scope (which in the case of the browser is the window object).

The second case is obviously a design flaw, but it's pretty easy to work around it by using closures.

Share Improve this answer Follow

answered Sep 25, 2008 at 15:38





In this case the inner this is bound to the global object instead of to the this variable of the outer function. It's the way the language is designed.



See "JavaScript: The Good Parts" by Douglas Crockford for a good explanation.



Share Improve this answer Follow











A this value is a special object which is related with the execution context. Therefore, it may be named as a context object (i.e. an object in which context the execution context is activated).



Any object may be used as this value of the context.

a this value is a property of the execution context, but not a property of the variable object.

This feature is very important, because in contrary to variables, this value never participates in identifier resolution process. I.e. when accessing this in a code, its value is taken directly from the execution context and without any scope chain lookup. The value of this is determinate only once when entering the context.

In the global context, a this value is the global object itself (that means, this value here equals to variable object)

In case of a function context, this value in every single function call may be different

Reference <u>Javascript-the-core</u> and <u>Chapter-3-this</u>

Share Improve this answer Follow

answered Mar 13, 2014 at 12:40



"In the global context, a this value is the global object itself (that means, this value here equals to variable object)". The global object is a part of the global execution context, as is the (es4) "variable object" and ES5 environment record. But they are different entities to the global object (e.g. an environment record can't be referenced directly, it is forbidden by the spec, but the global object can be). – RobG Apr 13, 2015 at 6:03



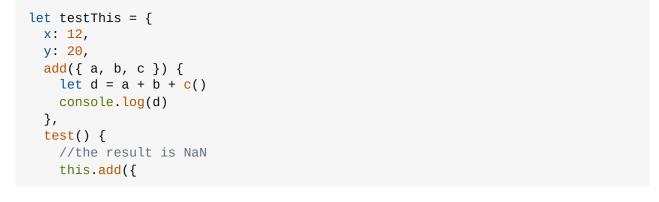
0

All the answers here are very helpful but I still had a hard time to figure out what this point to in my case, which involved object destructuring. So I would like to add one more answer using a simplified version of my code,









```
a: this.x,
      b: this.y,
      c: () => {
        //this here is testThis, NOT the object literal here
        return this.a + this.b
      },
    })
  },
  test2() {
    //64 as expected
    this.add({
      a: this.x,
      b: this.y,
      c: () => {
       return this.x + this.y
      },
    })
  },
  test3() {
    //NaN
    this.add({
      a: this.x,
      b: this.y,
      c: function () {
        //this here is the global object
        return this.x + this.y
      },
    })
  },
}
```

As here explained <u>Javascript - destructuring object - 'this' set to global or undefined</u>, <u>instead of object</u> it actually has nothing to do with object destructuring but how c() is called, but it is not easy to see through it here.

MDN says "arrow function expressions are best suited for non-method functions" but arrow function works here.

Share edited Sep 8, 2021 at 5:43 answered Sep 29, 2020 at 6:05

Improve this answer

Qiulang

12.3k • 19 • 96 • 150



this in JS:

There are 3 types of functions where this has a different meaning. They are best explained via example:



1. Constructor



// In a constructor function this refers to newly created object
// Every function can be a constructor function in JavaScript e.g.
function Dog(color){

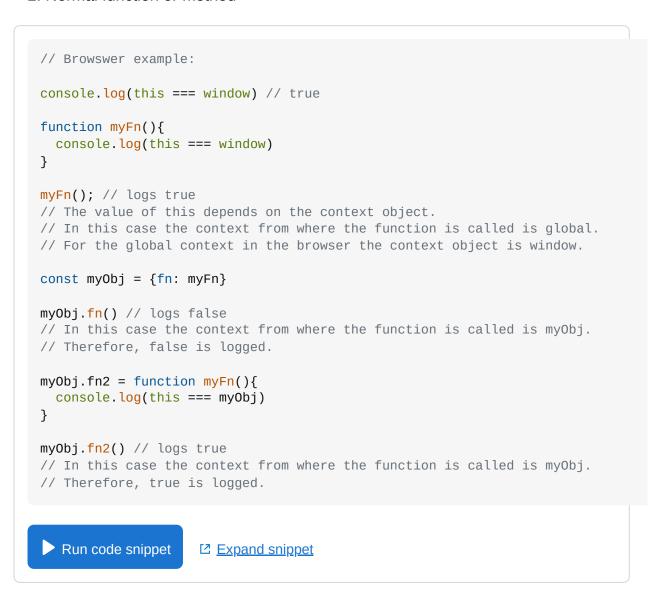
```
this.color = color;
}

// constructor functions are invoked by putting new in front of the function
call
const myDog = new Dog('red');

// logs Dog has color red
console.log('Dog has color ' + myDog.color);
Run code snippet

Expand snippet
```

2. Normal function or method



3. Event listener

Inside the function of an event handler this will refer to the DOM element which detected the event. See this question: Using this inside an event handler

Highly active question. Earn 10 reputation (not counting the association bonus) in order to answer this question. The reputation requirement helps protect this question from spam and non-answer activity.