O'REILLY®

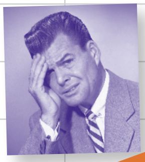
Head First JavaScript Programming

A Brain-Friendly Guide

A learner's guide to JavaScript programming

Watch out for common JavaScript traps and pitfalls

Launch your programming career in one chapter



Avoid embarrassing typing conversion mistakes

Extra

Bend your mind around 120 puzzles & exercises

Learn why everything your friends know about functions & objects is probably wrong

Eric Freeman & Elisabeth Robson

Head First JavaScript Programming Extra

by Elisabeth Robson

No objects were harmed in the making of this eBook.

Extra keeping track of this

this and function references

I try so hard to keep **this** happy, but every now and then, he just seems to change who he is completely. I don't know what I'm doing wrong...

Keeping track of `this` can be tricky. You're programming along, everything's going fine, and then, bam!; `this` doesn't behave at all like you think it should. You know `this` is supposed to be set to the object whose method you've called, but sometimes... well, `this` just isn't what you expect it to be. `this` is especially mysterious when you're calling a method outside of its ordinary context (the object it belongs to). We'll take a look at when that can happen, and ways to make sure `this` is set to exactly the object you want it to be in these situations.

Welcome to Webville Lounge.

We've got a DJ that knows how to play sounds:

```
var dj = {
    playsound: function() {
        console.log("Playing ", this.sound);
    },
    sound: "bells"
};
```

The DJ has just one method, playsound, which plays the sound that's stored in the sound property.

And we've got a controller that makes sure the DJ plays the right sound at the right time:

```
var controller = {
    start: function() {
        setInterval(dj.playsound, 1000);
    }
};
```

The controller also has just one method. The start method uses setInterval to call the DJ's playsound method every second so we get a repeating sound.

Test drive the DJ and controller

Let’s take the Webville Lounge for a spin and see some DJ action going on. Create a simple HTML file, add some code to start the `controller`, and see your music come to life:

```
<html>
<head>
<title>Webville Lounge</title>
<script>
var dj = {
    playsound: function() {
        console.log("Playing ", this.sound);
    },
    sound: "bells"
};
var controller = {
    start: function() {
        setInterval(dj.playsound, 1000);
    }
};
window.onload = function() {
    controller.start();
};
</script>
</head>
<body></body>
</html>
```

We’ve added the JavaScript to a basic HTML page.

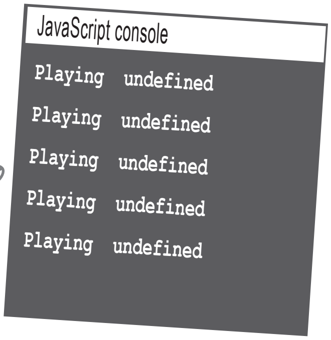
And we added the code to get the controller started once the page loads.

Cancel the concert; we’ve got a problem...

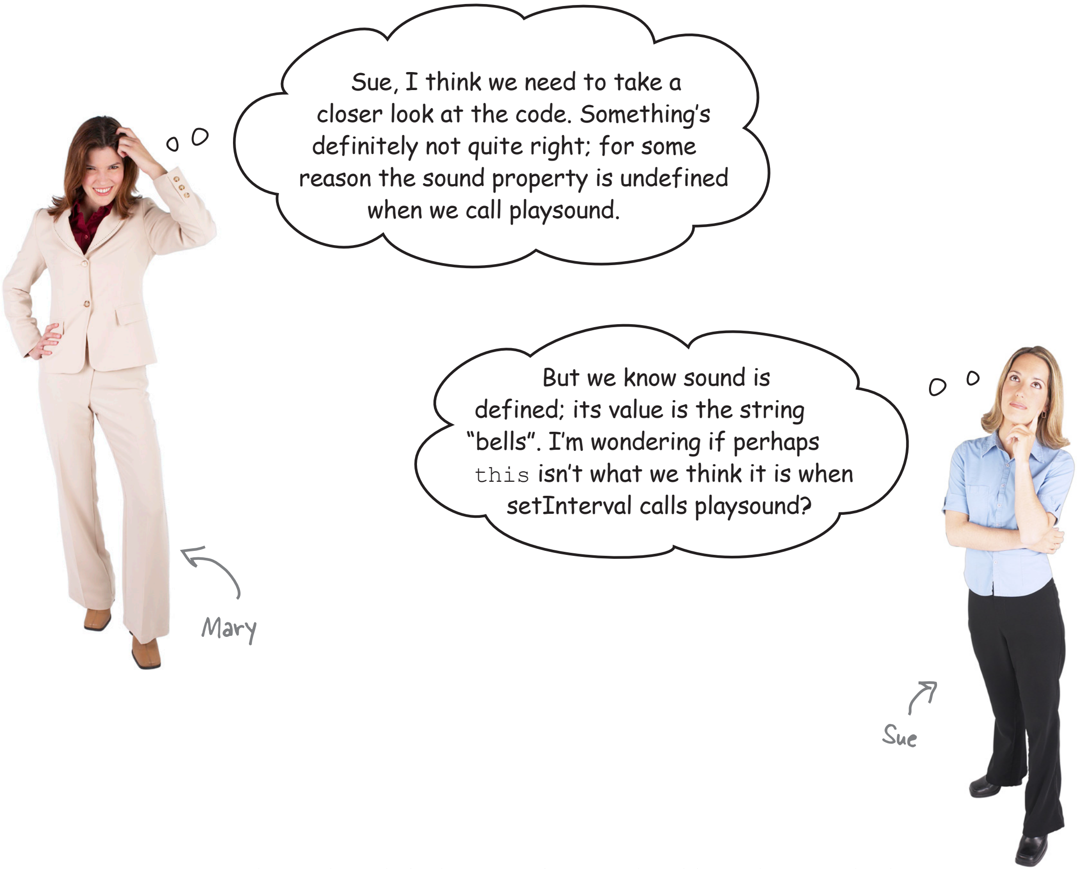
For some reason the `playsound` function isn’t playing the “bells” sound (or rather, in our simplified version of a DJ, displaying “bells” in the console).

What went wrong???

Hmm, it looks like the sound isn’t defined when we call the playsound method.



```
Playing  undefined
Playing  undefined
Playing  undefined
Playing  undefined
Playing  undefined
```


Sue, I think we need to take a closer look at the code. Something's definitely not quite right; for some reason the sound property is undefined when we call playsound.
But we know sound is defined; its value is the string "bells". I'm wondering if perhaps this isn't what we think it is when setInterval calls playsound?
Mary
Sue

Mary: Hmm. We know the rule is that `this` is set to the object whose method we're calling, right? And we're definitely calling the `playsound` method in the `dj` object.

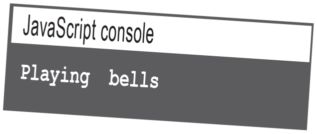
Sue: If you look more closely at the code, we're not actually calling `dj.playsound` ourselves. The `setInterval` function is doing that for us. We're just passing the `dj.playsound` method to `setInterval`.

Mary: True. But shouldn't the method call work in exactly the same way?

Sue: You'd think so, but I did some testing. I tried calling `dj.playsound` directly from the `window.onload` function and it worked fine. So there's something about the way we're passing the method to `setInterval` that's causing it not to work.

```
window.onload = function() {
    ~~controller.start();~~
    dj.playsound();
};
```

This works fine...



```
Playing bells
```

Mary: Interesting. Okay, well I think we need to take a closer look at what happens when we pass `dj.playsound` to `setInterval`. Clearly, we're missing something...

A closer look at the code...

Let's take a closer look at the code to see what it's doing, and maybe we can figure out what went wrong in the playsound method. (Make sure you read the flow of execution in the correct order, starting at 1.)

```
var dj = {
    playsound: function() {
        console.log("Playing ", this.sound);
    },
    sound: "bells"
};
var controller = {
    start: function() {
        setInterval(dj.playsound, 1000);
    }
};
window.onload = function() {
    controller.start();
};
```

3 We know that the playsound method is getting called (because we see "Playing..." over and over, but this.sound is undefined.

2 The start method calls setInterval, passing a reference to the method dj.playsound and a time interval, so dj.playsound will be called again and again every 1 second.

1 The first thing that happens after the page is loaded is we call the start method in the controller object.

This all seems straightforward. But look again at step 2: what, exactly, are we passing to setInterval when we pass dj.playsound? If you remember how setInterval (and setTimeout) work, you'll know that what we're passing is a reference to a function. But, what exactly is that reference in our case?

Check out Head First JavaScript Programming pages 192–193, page 409, all of Chapter 10 if you need a refresher.

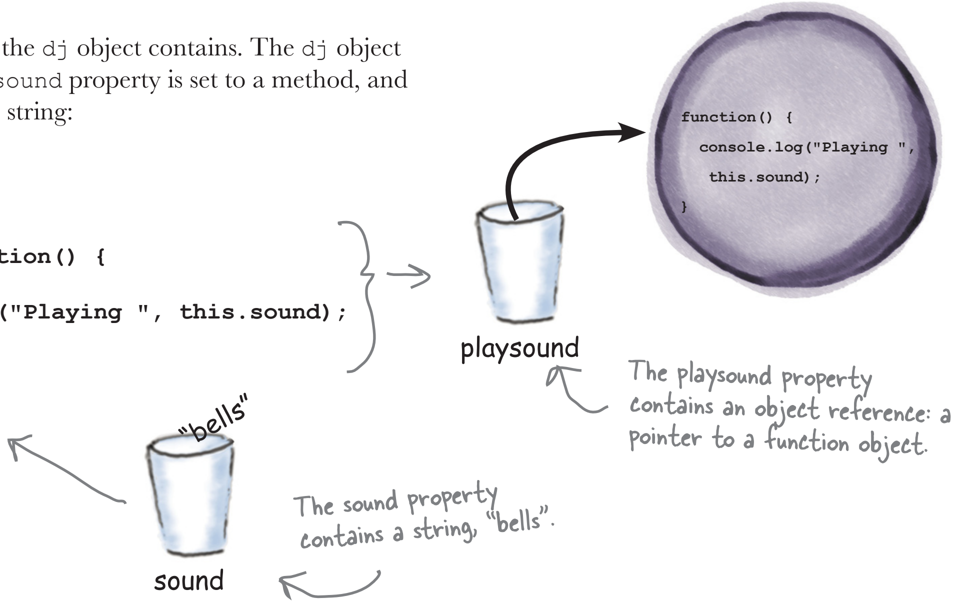
BRAIN POWER

Before turning the page, think about what dj.playsound is. Remember that in JavaScript, functions are objects. So we're actually passing a reference to an object—an object that happens to be a function. When setInterval calls that function, how will setInterval know that the function it's calling is actually a method in the dj object?

Function references

First, let's take a look at what the `dj` object contains. The `dj` object has two properties: the `playsound` property is set to a method, and the `sound` property is set to a string:

```
var dj = {
    playsound: function() {
        console.log("Playing ", this.sound);
    },
    sound: "bells"
};
```



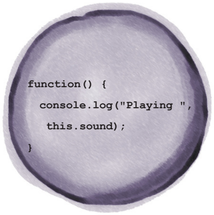
The playsound property contains an object reference: a pointer to a function object.

The sound property contains a string, "bells".

When we pass `dj.playsound` to `setInterval`, like this:

```
setInterval(dj.playsound, 1000);
```

what we're passing is what the `playsound` variable references, which is a function object:



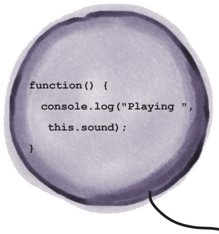
This is the value of dj.playsound. It's the thing that gets passed to setInterval.

Now, imagine that the implementation of `setInterval` looks something like this:

```
function setInterval(theFunction, milliseconds) {
   // after milliseconds has passed, call theFunction:
   theFunction();
}
```

Obviously, we don't know exactly how setInterval is implemented because it's internal to the browser's JavaScript engine, but we do know that at some point it calls the function you pass in.

So, what is `setInterval` calling? It's calling `playsound`, but without the `dj` object.



```
function() {
  console.log("Playing ",
   this.sound);
}
```

The value that gets assigned to theFunction parameter of setInterval (in our imagined implementation) is the value that's stored in the dj's playsound property.

```
function setInterval(theFunction, milliseconds) {
   // after milliseconds has passed, call theFunction:
   theFunction();
}
```

So when setInterval calls theFunction here...

... it's almost exactly like if we called the function playsound without the dj object, like this:

```
playsound();
```

Ah! Now I see the problem. setInterval is calling the method like a function. And because we're calling playsound as a function instead of as a method, `this` doesn't get set to the dj object.

For a refresher on how this works in method calls, check out Head First JavaScript Programming, pages 204-205.

You've got it.

Usually, when we call a method of an object, we call it like this:

```
dj.playsound();
```

When we call `playsound` as a method of the `dj` object, then `this` is correctly set to the `dj` object in the body of the `playsound` method, so everything works fine.

But here, `setInterval` is getting passed the right method, but isn't calling that method as a method; instead `setInterval` is calling it as a function, just as if you tried to call `playsound` like this:

```
playsound();
```

Without the "dj." in front of the call to `playsound`, there's no object to set `this` to.

So, what is `this` set to when `setInterval` calls `playsound`, if it's not set to the `dj` object? Good question. Let's find out...

What is this when setInterval calls the function?

We know that when `setInterval` calls the function we pass it, it's calling a function that looks like this:

```
function() {
    console.log("Playing ", this.sound);
}
```

This is what the playsound method looks like once it gets passed to setInterval. This happens behind the scenes of course, because we can't see inside setInterval.

And because `setInterval` is calling the function without the `dj` object (in other words, `setInterval` is calling the function as a function, not as a method), the `this` in the body of `playsound` doesn't get set to the `dj` object.

So what is `this` set to in `playsound`? Is it `undefined`? Or set to something else? We can find out by adding a line of code to display the value of `this` when `playsound` is called:

```
playsound: function() {
    console.log("(playsound) This is: ", this);
    console.log("Playing ", this.sound);
}
```

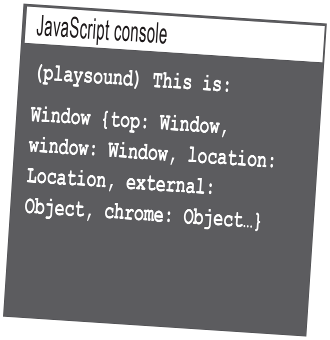
We're adding this line of code so we can see what this is set to when the function is called by setInterval.

Go ahead and add this line to your code and let's see what the value of `this` is in the `playsound` method when it's called by `setInterval`.

A quick test drive...

Now, when we run the code, we can see that `this` in the `playsound` method is set to the `window` object. `window` is the default value for `this` in your code. Because `setInterval` is calling `playsound` as a function rather than as a method of the `dj` object, the value of `this` isn't changed from the `window` object to another object (like it is when you call a method of an object).

So now the question is: how do we make sure that `setInterval` calls `playsound` as a method of the `dj` object instead of as a function?



JavaScript console

```
(playsound) This is:
Window {top: Window,
window: Window, location:
Location, external:
Object, chrome: Object…}
```

Sharpen your pencil

To see that **this** is set to the window object in a regular function call, try running this code in the console (you can just copy and paste the code into your browser console):

```
function testThis() {
    console.log("This is: ", this);
}
testThis();
```

Making sure this gets set correctly when the playsound method is called by setInterval

There are a couple of different ways we can make sure that `this` is set to the correct object when `playsound` is called by `setInterval`. We'll step through both.

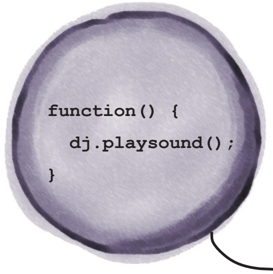
The first is straightforward. We know that the easiest way to get `this` set to the correct object is to call `playsound` as a method of `dj`. So, what if we pass a simple anonymous function to `setInterval` that does exactly that? Let's see how that might work.

First, we'll change the call to `setInterval` like this:

```
setInterval(function() { dj.playsound(); }, 1000);
```

Don't forget to add () after dj.playsound! We really do want to call the method this time.

Now when we call `setInterval`, we pass the anonymous function, which `setInterval` calls every 1 second:

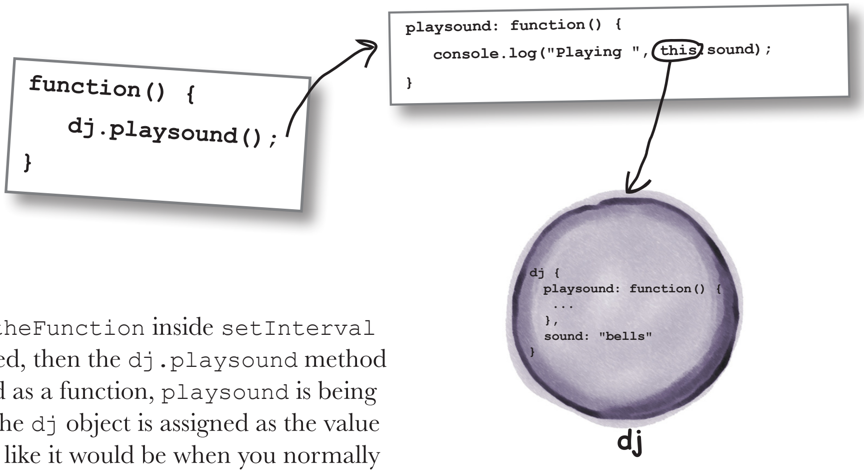


```
function() {
  dj.playsound();
}
```

The value that gets assigned to theFunction parameter of setInterval (in our imagined implementation) is the anonymous function that calls dj.playsound.

```
function setInterval(theFunction, milliseconds) {
   // after milliseconds has passed, call theFunction:
   theFunction();
}
```

When setInterval calls theFunction, it's calling the anonymous function we passed in, which then calls dj.playsound, like this:



```
function() {
   dj.playsound();
}
```

```
playsound: function() {
   console.log("Playing ", this.sound);
}
```

```
dj {
  playsound: function() {
    ...
  },
  sound: "bells"
}
```

When the anonymous function (named `theFunction` inside `setInterval` in our imaginary implementation) is called, then the `dj.playsound` method is called. But now, instead of being called as a function, `playsound` is being called as a method of the `dj` object. So the `dj` object is assigned as the value of `this` in the body of `playsound`, just like it would be when you normally call a method of an object.

Test drive the new controller code

Let's give the code a try and see if it fixes our music controller. Make sure you've made the updates to the code, like this:

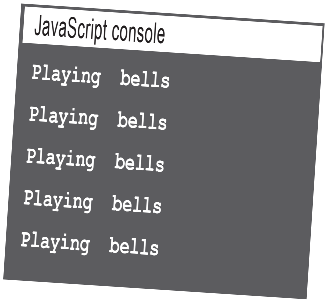
```
var dj = {
    playsound: function() {
        console.log("Playing ", this.sound);
    },
    sound: "bells"
};
var controller = {
    start: function() {
        setInterval(function() { dj.playsound(); }, 1000);
    }
};
window.onload = function() {
    controller.start();
};
```

We've removed the extra console.log line that we were using to display the value of this; guess we're pretty confident the new code will work!

And we've updated the call to setInterval to pass the anonymous function as the first argument.

Don't forget the ().

And, when you load the page, you should see your DJ object working just like it should, displaying the "bells" sound every 1 second.



JavaScript console

```
Playing  bells
Playing  bells
Playing  bells
Playing  bells
Playing  bells
```

there are no Dumb Questions

Q: There's really no way for setInterval to figure out that playsound is a method of the dj object? It seems like setInterval ought to be able to figure that out from the name "dj.playsound".

A: No, setInterval really can't figure that out. To setInterval, playsound looks like just a regular function that's disconnected from any particular object. There's nothing in the function object that says "I belong to the dj object". The fact that we use "dj." in "dj.playsound" when we pass the function doesn't mean the function object has any information about the dj object in it.

Q: Remind me how to stop the interval timer?

A: For now, just close the browser window to stop the code running. Remember that setInterval returns a timer object you can save in a variable. To stop the timer, you can pass it to the clearInterval function. We'll improve the controller code to add a stop method that does this shortly.

Using bind to set the value of this

Another way to make sure that this has the correct value when playsound is called from setInterval is to set the value yourself using bind. bind is a method you can use on any function. You pass bind an object that you want to use as this in the body of that function.

Now, if you've read *Head First JavaScript Programming*, you might think that sounds a bit like the call method. But there's an important difference. With call, we specify the object to use for this in the function we're calling, and that function gets called *right away*.

For a refresher on call, read Chapter 13 in Head First JavaScript Programming.

With bind, the function doesn't get called; instead, a *new function* is returned. The new function is exactly like the original one, except that the value of this in the new function is *bound* (set) to the object you specified in bind. Let's take a look at an example to compare call and bind.

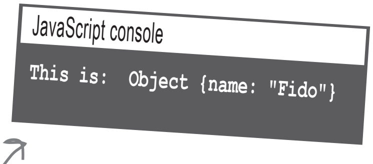
Let's modify the example from page 10, testThis, to use call, like this:

```
function testThis() {
    console.log("This is: ", this);
}
var dog = {
    name: "Fido"
};
testThis.call(dog);
```

We've added a variable dog, that is an object with one property name.

We can specify that we want to use the dog object as the value for this in the body of testThis by using call, and passing the dog object.

This calls testThis right away, and we see dog in the console as the value of this.



JavaScript console

```
This is:  Object {name: "Fido"}
```

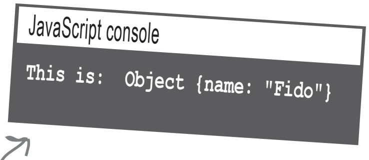
Remember, if we don't specify a value for this in testThis, the value defaults to the window object.

Now, change the code to use bind instead:

```
function testThis() {
    console.log("This is: ", this);
}
var dog = {
    name: "Fido"
};

var newFunction = testThis.bind(dog);
newFunction();
```

Now we're using the bind method, and passing dog. testThis doesn't get called at this point; instead bind returns a new function with this bound to dog.

To call testThis, we now have to call the function that was returned from bind, newFunction.



JavaScript console

```
This is:  Object {name: "Fido"}
```

When we call newFunction, we get the same result as above.

How does that help us? In our setInterval example, we don't want to call the function; we want to pass it to setInterval.

That's exactly why we're going to use bind.

You're right; we don't want to call `dj.playsound`; we want `setInterval` to do that. But we want `setInterval` to call `playsound` with the `dj` object assigned to `this`. In other words, we want to pass `setInterval` a function in which `this` is *bound* to the `dj` object.

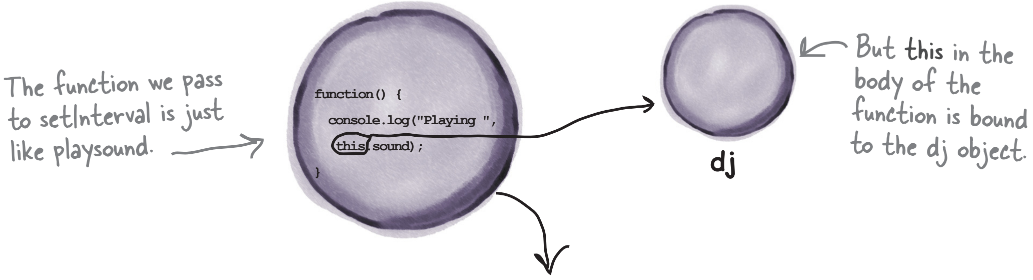
We can create a new function that is exactly like `playsound` with `this` bound to the `dj` object like this:

```
dj.playsound.bind(dj)
```

We're calling `bind` on the `dj.playsound` method, and passing the `dj` object to bind to `this`. It looks a bit weird, but that's exactly what we need to pass to `setInterval`:

```
var newPlaysound = dj.playsound.bind(dj);

setInterval(newPlaysound, 1000);
```

Now what we're passing to `setInterval` is a reference to a function in which `this` is bound to to the `dj` object:



The function we pass to setInterval is just like playsound.

```
function() {
  console.log("Playing ",
   this.sound);
}
```

But this in the body of the function is bound to the dj object.

```
function setInterval(theFunction, milliseconds) {
   // after milliseconds has passed, call theFunction:
   theFunction();
}
```

So when setInterval calls the function, it works fine because this is bound to the correct object.

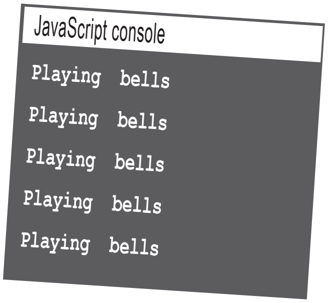
Test drive the controller with bind

Once again, update your code and reload the page and let's see if our new solution using bind works:

```
var dj = {
    playsound: function() {
        console.log("Playing ", this.sound);
    },
    sound: "bells"
};
var controller = {
    start: function() {
        setInterval(dj.playsound.bind(dj), 1000);
    }
};
window.onload = function() {
    controller.start();
};
```

Notice, we've combined the two lines on the previous page into one by passing the result of the bind method call directly into setInterval.

And our new code works perfectly: we see the "Playing bells" message in the console, which means this is correctly bound to the dj object when playsound is called from setInterval.



```
Playing  bells
Playing  bells
Playing  bells
Playing  bells
Playing  bells
```

there are no Dumb Questions

Q: I remember from Head First JavaScript Programming that we could pass arguments to the function we were calling with the call method. Can we pass arguments along with bind?

A: Yes, you can. Just as with call, any additional arguments you pass to bind are passed as arguments to the function when it's called. So if you changed the playsound method to take one argument, say the volume to play the sound, you'd use bind like this:

```
dj.playsound.bind(dj, "loudly")
```

When setInterval calls playsound, it will pass "loudly" along as an argument.

Q: Which solution is better: using an anonymous function to wrap a call to dj.playsound, or using bind?

A: Neither is better, and in this situation, they do exactly the same thing: allow you to bind the dj object to this in the body of playsound. In both solutions, you're creating a new function.

In some situations, you'll find one of these solutions is more suited than the other, but in this case, either one works fine.

Adding start and stop buttons to Webville Lounge

At this point, you're probably sick of having to close the browser window to get your DJ to stop playing the bells, so let's add both a start and stop button to the page so you have more control. The start button will call `controller.start` to start the music, and we'll add a new `stop` method to the controller that the stop button will call to stop the music.

Begin by updating your HTML to add the two buttons, start and stop:

```
<html>
<head>
<title>Webville Lounge</title>
<script>
   // JavaScript code here...
</script>
</head>
<body>
    <button id="start">start</button>
    <button id="stop">stop</button>
</body>
</html>
```

If you need a refresher on setting up click handlers for form elements like buttons, check out pages 358–359 in Head First JavaScript Programming.

Next, we'll add code to the `window.onload` handler to add click handlers to both buttons. We'll also remove the code to call `controller.start` from `window.onload`, because now we'll call this method when we click on the start button.

```
window.onload = function() {
    ~~controller.start();~~

    var startButton = document.getElementById("start");
    startButton.onclick = controller.start;

    var stopButton = document.getElementById("stop");
    stopButton.onclick = controller.stop;
};
```

We're getting the button element objects from the DOM using their ids, "start" and "stop" respectively.

Notice that we're using methods in the controller as our click handlers! This is totally fine because as long as what we're assigning to the onclick property of the button is a function reference, the button will call that function when you click on the button.

This might seem weird, but it's really the same thing as defining a function at the top level and assigning that function to the onclick property, like we do on page 359 of Head First JavaScript Programming. In both cases, we're assigning a function reference to the onclick property: a function to call when the click event occurs.

Finally, we need to modify the controller a bit. We'll add a new property, `timer`, that will store the timer we create in the `start` method; modify the `start` method so we save the interval timer we're creating; and add a new method, `stop`, that will clear the interval timer:

```
var controller = {
    timer: null,
    start: function() {
        this.timer = setInterval(dj.playsound.bind(dj), 1000);
    },
    stop: function() {
        clearInterval(this.timer);
    }
};
```

We add a property to save the timer, and use it to store the timer created in the start method.

To stop the timer, we simply pass it to the clearInterval function.

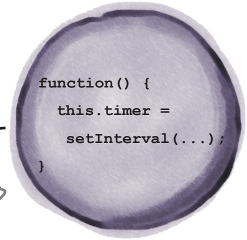
Okay, that should do it! Let's reload the page and...

Wait just a moment. I think we're going to have exactly the same problem we had before, aren't we?

Great catch; yes we are.

We've got a different situation, but the problem is basically the same. We're referencing a method in an object, and storing that function in the `onclick` property of a button:

```
startButton.onclick = controller.start;
```



```
function() {
  this.timer =
   setInterval(...);
}
```

What gets assigned to the onclick property is a function reference to the start function. Just like before, the start function has no information about the controller object in which it's defined.

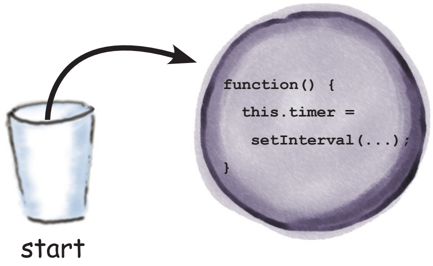
When you click on the button, and the click handler function is called, it'll be called as a function, not as a method.

Once again, we're losing the correct binding for `this` in the body of our method; this time, in the method we're calling as the click handler—that is, in our `start` method (and likewise for the `stop` method).

The click handler problem up close

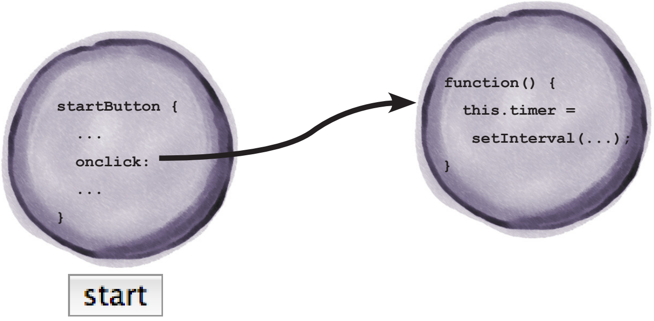
1. First, we get a reference to the controller.start method:

```
startButton.onclick = controller.start;
```

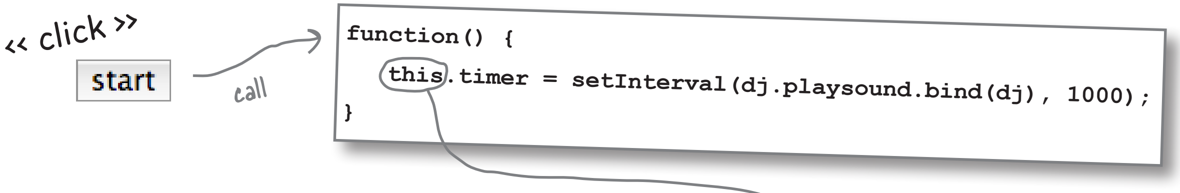


2. Then, we assign that reference to the onclick property of the start button object:

```
startButton.onclick = controller.start;
```



3. You click on the start button, which causes the button to call the function referenced in its onclick property:



```
function() {
   this.timer = setInterval(dj.playsound.bind(dj), 1000);
}
```

4. The start method is called as a function, so the controller object is not bound to this in the body of the function.

? this is not bound to the controller object... but what is it bound to? The window object?

What is the value of this in a click handler?

We're pretty darn sure that `this` will not be bound to the `controller` object in the `start` function when you click the button. But what is `this` bound to in this case? Is it the `window` object like before (since `window` is the default value for `this`), or is it something else? Let's do a litle more testing to find out.

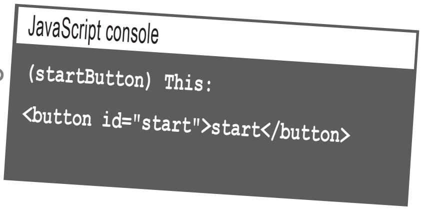
Temporarily change your code to set the `startButton`'s `onclick` property to a function that simply displays the value of `this` in the console, like this:

```
window.onload = function() {
    var startButton = document.getElementById("start");
    ~~startButton.onclick = controller.start;~~

    startButton.onclick = function() {
        console.log("(startButton) This: ", this);
    };

    var stopButton = document.getElementById("stop");
    stopButton.onclick = controller.stop;
};
```

All we've done is temporarily remove the line that sets the click handler property of the button to the controller.start method, and instead, we're setting it to an anonymous function that displays the value of this in the console.

Make sure you've updated your code (including adding the `stop` method to the `controller` like we showed a couple of pages ago). Reload the page and take a look at the output in the console:

Here's what we see in the console (Chrome).



JavaScript console

```
(startButton) This:
<button id="start">start</button>
```

Interesting! It looks like the default value for `this` in the `startButton`'s click handler is the `startButton` object (note that Chrome displays this object using HTML, rather than JavaScript).

That's actually the case for all DOM click handlers. That is, the default value for `this` in any click handler is the object whose click handler you called. In other words, it works just like a regular method call. You've called the button's `onclick` method, so the value of `this` in that method is bound to the button object.

Exercise

Before you turn the page to see our solution, try to fix the code using one of the solutions we used before so that `this` is bound to the controller object in the start function when you click the button.

Fixing the start and stop buttons

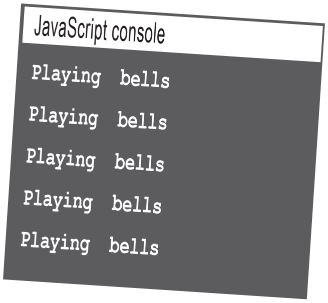
While it's often handy to have `this` refer to the object you clicked in the click handler function, in this particular case, we really want `this` to refer to the `controller` object in both the `start` and `stop` methods. We can fix the code in a couple of different ways, just as you saw before. We can either wrap the calls to the respective methods in anonymous functions, or we can use `bind`. We're going to go with using anonymous functions this time (since we used `bind` before). Again, in this case, it doesn't matter which solution you choose as both accomplish the same thing: they both set `this` to the `controller` object in the `start` and `stop` methods.

```
window.onload = function() {
    var startButton = document.getElementById("start");
    startButton.onclick = function() {
        controller.start();
    };

    var stopButton = document.getElementById("stop");
    stopButton.onclick = function() {
        controller.stop();
    };
};
```

Don't forget the ().

The `controller` code doesn't change (from page 17). Make these changes, reload the page and give the buttons a try!

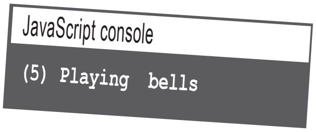
Now, when you click on the start button, you'll see bells playing every 1 second...



```
Playing bells
Playing bells
Playing bells
Playing bells
Playing bells
```

...and when you click stop, you'll see the bells stop playing. Success!!

Depending on your browser, you might see the results displayed like this instead. This just means "repeat this line 5 times."



```
(5) Playing bells
```

Well done!

Not only have you solved the mystery of what happens to `this` in two situations: passing a method to `setInterval` (and `setTimeout` too!) and using a method as a click handler function; you've also learned how to use `bind`.

That's a lot for one project, so sit back, relax, put on some good music and give yourself a good pat on the back.

Thanks for helping us out! Webville Lounge couldn't have had the concert without you...

The complete code

Below you'll find the complete code for our solution. You can also find it online at https://github.com/bethrobson/Head-First-JavaScript-Programming/tree/master/extras in the file timer.html, and a link to the project at http://wickedlysmart.com.

```
<html>
<head>
<title>Webville Lounge</title>
<script>
var dj = {
    playsound: function() {
        console.log("Playing ", this.sound);
    },
    sound: "bells"
};

var controller = {
    timer: null,
    start: function() {
        this.timer = setInterval(dj.playsound.bind(dj), 1000);
    },
    stop: function() {
        clearInterval(this.timer);
    }
};

window.onload = function() {
    var startButton = document.getElementById("start");
    startButton.onclick = function() {
        controller.start();
    };
    var stopButton = document.getElementById("stop");
    stopButton.onclick = function() {
        controller.stop();
    };
};

</script>
</head>
<body>
    <button id="start">start</button>
    <button id="stop">stop</button>
</body>
</html>
```

You mean we're done? Aren't you going to show us how to play a real sound when we click start? That would be much more exciting...

We agree!

But that's a whole 'nother project. Stay tuned at wickedlysmart.com for more music... coming soon.

In the meantime, practice keeping track of `this` by working through the projects in *Head First JavaScript Programming* again and make sure you know what `this` is bound to in all those examples.

Or invent a few examples of your own! Let us know what you discover.