How being stumped by one line of code helped me understand context in JavaScript

I'm currently enrolled in *Course 220: Front End Development with JavaScript* at Launch School. While preparing for the assessment for this course, I was reviewing the projects I had completed and was struck by one assignment in particular in which we were given an unusual code structure. Here is a simplified version of it:

On line 1 we declare the inventory variable within the global scope. This way we can access it in the console as we are developing. Then we use an Immediately-Invoked Function Expression (IIFE) to add properties to the inventory object. We use an IIFE so that the properties are private and encapsulated in the inventory object.

The final line is where I got stumped.

```
$(inventory.init.bind(inventory));
```

I stared at that line a long time.

I wondered, why is it necessary to bind the `inventory.init` method to the `inventory` object context? And why is this line wrapped in a jQuery function?

Since it took me a long time and a lot of reading to answer these questions for myself, I thought I would share how I broke down what was happening and what I learned about context in JavaScript and the jQuery function in the process.

Let's start with the first question:

Why is it necessary to bind the inventory.init method to the inventoryobject context?

In my understanding of context in JavaScript so far, I knew that if you call a method on an object the context is set to the object calling it. So, I thought, couldn't you just write `$(inventory.init)` ?

I tried that and got this error:

```
Uncaught TypeError: this.bindEvents is not a function
```

What? But `bindEvents` is defined in the `inventory` object, and the `inventory` object is `this` right? ... Wrong. After using a breakpoint in the console I discovered that `this` was pointing to the `Window` , the implicit global context. But how did that happen? Why was the `inventory` context lost?

Then came my first a-ha moment. I realized that the `inventory.init` method was not being immediately called. If it was being called it would be followed by a parenthesis like so: `inventory.init()` .

And unlike `call` or `apply` , `bind` does not call the function. Instead, its purpose is to permanently bind the context to the object passed to it.

This is when I remembered an important concept: **function execution context is determined not when the function is defined but instead when it is *executed*.**

For example:

```
var foo = {
  bar: function() {
    return this;
  }
};

var baz = foo.bar;
```

```
foo.bar(); // => returns Object { ... }
baz();     // => returns Window { ... }
```

`foo.bar()` sets `this` to foo because it is method invocation. However, when we assign `foo.bar` to `baz` and then later call `baz()`, the `foo` context is lost because we are using function invocation, so the implicit `Window` context is used as `this` instead.

To correct this, we would use `bind` like so:

```
var foo = {
  bar: function() {
    return this;
  }
}

var baz = foo.bar.bind(foo);

foo.bar(); // => returns Object { ... }
baz();     // => returns Object { ... }
```

With the first piece of the puzzle solved, it was time to move to the next question:

Why is the line wrapped in a jQuery function?

Here's the line in question again for our reference:

```
$(inventory.init.bind(inventory));
```

The jQuery function, which the `$` is a shorthand for, was the other mysterious element to me. Couldn't we leave it out and just write:

```
inventory.init()
```

And anyway, I'm used to seeing the jQuery function wrapping selectors like `$('p')`, what is it doing wrapping a function?

That's when I remembered that a shortcut for writing

```
$(document).ready(function() { } )
```

is

```
$(function() { } );
```

Then I realized that what we were doing was instead of using an anonymous function like the line above, we were telling jQuery to run the `inventory.init` function after the document was ready.

Because the jQuery function was not immediately calling the `inventory.init` function, but instead storing it to be called later when the document was ready, the context is lost if you don't `bind` the context to the inventory object.

This is because when the function is actually called it is being called by the jQuery function, making it function invocation, so the implicit function context of `Window` is used. This is why we instead write `$(inventory.init.bind(inventory))` to permanently bind the function context to the `inventory` object so that when the jQuery function later calls it the context is retained.

This also helps explain why on line 10,

```
$("#add_item").on("click", this.newItem.bind(this));
```

we have to bind the context again, because we are not immediately calling the `this.newItem` method, but instead jQuery will call it later using function invocation when the click event fires.

So, in summary, if we were to explain what happens in the last line of code,

```
$(inventory.init.bind(inventory));
```

we would say that we bind the `inventory.init` method to the inventory object context and pass it to the jQuery function as a callback that will be called as function invocation after the document has loaded and the DOM is ready.

Phew! Now I finally understood what was happening on that final line of code and felt I had a better understanding of context in JavaScript and how the jQuery function works.

For further reading, I highly recommend Gentle explanation of 'this' keyword in JavaScript.

Note: Since I'm relatively new to these concepts, it's very possible there are mistakes in this post that need corrected. Please help me out and add corrections in the comments and I will edit this post.