We're teaching a special *live* (and free) 6-week intro to programming and computer science course.
[Apply before it's too late.](#)

How to Think About "this" in JavaScript

by Gordon Zhu, May 2023

[If you prefer video, see the [YouTube version of this post](#).]

In 2013, I found myself on a team that was building one of the largest AngularJS applications in the world at the time. And there was no way to avoid `this`—it was everywhere.

Now that I had no choice, I finally decided to tackle `this` once and for all—my previous hazy understanding would no longer cut it. I wanted a foolproof process that would allow me to methodically analyze any usage of `this` with pinpoint precision. I had no patience for overly complicated processes that included irrelevant details (this was my experience with documentation and tutorials).

After several long sessions, I designed my own system and immediately put it to use. Overnight, it changed `this` from a dreadful experience into something that was satisfyingly predictable.

I'll demonstrate how the system works by example. You'll see examples with standard function calls, arrow functions, methods, `class`, `bind`, `call`, `apply`, `new`, `super`, and even `super.method()`. From studying these examples, you'll learn how to figure out how `this` works in any situation you encounter.

The heart of the system is this notably compact flowchart. For now, I want you to appreciate its compactness. Don't worry about the details yet, we'll get into that in the examples. The point is that `this` may be complicated, but it's not *that* complicated. You can get a hold of it if you're careful and use a systematic approach.

```
┌──────────────────────────────┐
│  _____ _   _ ___ ____        │
│ |_   _| | | |_ _/ ___|       │
│   | | | |_| || |\___ \       │
│   | | |  _  || | ___) |      │
│   |_| |_| |_|___|____/       │
└──────────────────────────────┘
```

```
╔═════════════════════════════════════════════════════╗
║                    SPECIAL CASE                     ║
╚═════════════════════════════════════════════════════╝

┌──────────┐   ┌──────────┐   ┌──────────┐   ┌──────────┐
│bind, call,│  │   new    │   │    =>    │   │  super   │
│  apply   │   │          │   │          │   │ .method()│
└──────────┘   └──────────┘   └──────────┘   └──────────┘
     │              │              │              │
     │              │              │              │
     ▼              ▼              ▼              ▼
┌──────────┐   ┌──────────┐  ┌────────────┐ ┌────────────┐
│ Up to you│   │ New empty│  │Outside `this`│ │Outside `this`│
│          │   │  object  │  │when function│ │when function│
└──────────┘   └──────────┘  │ is created │ │ is invoked │
                             └────────────┘ └────────────┘

╔═════════════════════════════════════════════════════╗
║                     COMMON CASE                     ║
╚═════════════════════════════════════════════════════╝

                  ┌──────────┐          ┌──────────┐
                  │   In a   │          │          │
                  │ function?│───No───▶│  window  │
                  └──────────┘          └──────────┘
                        │
                       Yes
                        │
                        ▼
                  ┌──────────┐
                  │Invoked as│
                  │  method? │
                  └──────────┘
                        │
             ┌────Yes───┴───No────┐
             │                    │
             ▼                    ▼
     ┌──────────────┐    ┌────────────────┐
     │              │    │ window (undefined│
     │  Left of dot │    │if function created│
     │              │    │    on class)   │
     └──────────────┘    └────────────────┘
```

The flowchart is broken up into two parts. At the top, you have the "SPECIAL CASE". These cases are easily identifiable by specific keywords and language constructs.

If no special case applies, then move to the "COMMON CASE". Analyzing `this` in the common case boils down to just two questions: are we "in a function?" and if so, is that function "invoked as method?".

As you can see, technically, this is very simple. The only prerequisites are (1) basic logical reasoning skills and (2) basic programming skills. But as you'll soon find, bringing everything together is easier said than done.

As you study the examples, your attention should be on my thought-process. If you pay close attention to my *reasoning* with each example, you should be able to figure out any novel situation on your own.

[Common Case] Example 1 of 5

```
console.log(this); // this?
```

- Decision: Special case? No, so we examine the common case.
- Decision: In a function? No.
- Result: `this` equals `window`.

[Common Case] Example 2 of 5

```
function logThis() {
  console.log(this); // this?
}

logThis();
```

- Decision: Special case? No, so we examine the common case.
- Decision: In a function? Yes, we're in the body of `logThis`.
- Decision: Invoked as method? No, `logThis` is not invoked as a method.
- Result: `this` equals `window` (not `undefined` because `logThis` was not created on a class).

[Common Case] Example 3 of 5

```
class MyClass {
  static method() {
    console.log(this); // this?
  }
}

MyClass.method();
```

- Decision: Special case? No, so we examine the common case.
- Decision: In a function? Yes, we're in the body of `MyClass.method`.
- Decision: Invoked as method? Yes (`MyClass.method()`).
- Result: `this` equals `MyClass` (in the invocation, `MyClass` is to the left of the dot).

I love this example because the code *seems* complicated (due to `class` and `static`). But if you follow the flowchart, you'll cut to the heart of the matter with zero bullshit.

[Common Case] Example 4 of 5

```
class MyClass {
  static method() {
    console.log(this); // this?
  }
}

const test = MyClass.method;
test();
```

Here we have the same class from the previous example, but we're calling the function a little differently. Instead of invoking the function as a method, we're extracting it into a variable called `test`, and then we're invoking `test`.

- Decision: Special case? No, so we examine the common case.
- Decision: In a function? Yes, we're in the body of `MyClass.method`.
- Decision: Invoked as method? No! (`test()`).
- Result: `this` equals `undefined` because `MyClass.method` was created on a class.

[Common Case] Example 5 of 5

```
const myObject = {
  method: function () {
    console.log(this); // this?
  }
}

function outer(input) {
  input(); // <-- invocation
}

outer(myObject.method);
```

To start, we need to figure out where the function is actually invoked. It's not obvious at first glance. After inspecting the `outer` function, we see that `myObject.method` is invoked with the expression `input()`.

Now we have everything we need to know to do our analysis. Note that everything we've done so far has absolutely nothing to do with `this`. We've only analyzed basic mechanics, variables, objects, and functions. So a crisp and clear fluency with the basic mechanics is an absolute prerequisite to figuring out `this`.

I felt compelled to add this note because too often people think they're confused about `this` or some other complicated issue, when the real problem is basic fluency with programming.

- Decision: Special case? No, so we examine the common case.
- Decision: In a function? Yes, we're in the body of `myObject.method`.
- Decision: Invoked as method? No! (See `// <-- invocation` comment).
- Result: `this` equals `window` (not `undefined` because `myObject.method` was not created on a class).

What about callback functions?

You might be wondering, wait, wasn't that a callback function in the previous example? And the answer is yes. I could have said my `myObject.method` is passed into `outer` as a callback function, but I didn't, because it doesn't add anything to the analysis. It's an irrelevant detail.

The key thing is determining how `myObject.method` function is actually invoked. There's nothing special about callback functions. If there were, I'd have to incorporate them into the flowchart. A common problem is that many instructors introduce callback functions within the context of built-in functions.

This makes things really confusing because you can't easily see the source code for built-in functions, and so if you pass a function into a built-in function, you can't see how your function is invoked, which is key to determining `this`. Basically built-in functions force you to reason about what's happening with incomplete information.

It doesn't make sense to rush into situations with built-in functions until you can confidently handle situations where you do have the full picture. That's why I'll give special treatment to built-in functions at the very end of this post. In that section, I'll also talk about a really damaging yet common misconception held by many experienced programmers when it comes to built-in functions and `this`.

[Special Case]: `bind`, `call`, `apply` (example 1 of 1)

```
function logThis() {
  console.log(this); // this?
  // Note that this function will be invoked 3 times.
}

const gordon = {name: 'Gordon'};

logThis.bind(gordon)();
logThis.call(gordon);
logThis.apply(gordon);
```

- Decision: Special case? Yes. We can easily tell because the function is invoked with `bind`, `call`, or `apply` keywords.
- Result: `this` equals `gordon`. Each time `logThis` is invoked (a total of 3 times), we explicitly set the `this` value to `gordon`.

The key idea is that `bind`, `call`, and `apply` allow you to control the `this` value of a function.

[Special Case]: `new` (example 1 of 4)

```
function Person(name) {
  this.name = name; // this?
}

const p = new Person('Gordon');
```

- Decision: Special case? Yes. We can tell because the function is invoked with `new`.
- Result: `this` equals a new empty object. and you can set properties to new object using this

When you invoke a function with `new`, JavaScript will set `this` to a new empty object. If you want, you can optionally attach properties to the object (like `name`), and then in the background, JavaScript will implicitly return `this`.

[Special Case]: `new` (example 2 of 4)

```
class Person {
  constructor(name) {
    this.name = name; // this?
  }
}

const p = new Person('Gordon');
```

- Decision: Special case? Yes. We can tell because `Person` is invoked with `new`.
- Result: `this` equals a new empty object.

[Special Case]: `new` (example 3 of 4)

```
class Person {
  constructor(name) {
    this.name = name;
  }

  thisOutside = this; // this?
}

const p = new Person('Gordon');
p.thisOutside === p; // true
```

- Decision: Special case? Yes. We can tell because the `Person` function is invoked with `new`.
- Result: `this` equals a new empty object. This example shows that `this` will be the same new empty object *everywhere* inside of the class body, both inside and outside of the constructor.

[Special Case]: `new` (example 4 of 4)

```
class Parent {
  constructor() {
    // P1
  }

  // P2
}

class Child extends Parent {
  constructor() {
    super();
    // C1
  }

  // C2
}

const child = new Child();
```

- Decision: Special case? Yes. We can tell because `Person` is invoked with `new`.
- Result: `this` equals a new empty object. `this` will be the same new empty object *everywhere* inside of both `Child` *and* `Parent` (since `Parent` is invoked via `super()`). That means that `this` will be the same empty object at all points: "P1", "P2", "C1", and "C2".

It's important not to confuse `super()` with `super.method()`, which is a completely *different* special case (we'll see an example of this later on).

Special Case: Arrow functions `=>` (example 1 of 1)

```
const o = {
  arrowMethod: () => {
    console.log(this); // this?
  }
};
```

- Decision: Special case? Yes. We can tell because we have the syntax for an arrow function `=>`.
- Intermediate Result: `this` equals outside `this` when function (`o.arrowMethod`) is created.

To determine the `this` value outside of `o.arrowMethod`, we repeat the process:

- Decision: Special case? No. Outside of the arrow function, no special cases apply.
- Decision: In a function? No.
- Final Result: `this` equals `window`.

An important implication of arrow functions is that at creation, `this` is set once and then it never changes—it's permanent. As long as you call the same arrow function, `this` will always be the same.

Special Case: `super.method()` (example 1 of 2)

```
class Parent {
  static parentMethod() {
    console.log(this); // this?
  }
}

class Child extends Parent {
  static childMethod() { context of final this
    super.parentMethod();
  }
}

Child.childMethod();
```

- Decision: Special case? Yes. `parentMethod` is called as a method on super, which falls into the `super.method()` case.
- Intermediate Result: `this` equals outside `this` when function (`parentMethod`) is *invoked*.

Since `parentMethod` is invoked inside of `childMethod`, we must determine the `this` value inside of `childMethod`. To answer this question, we repeat the process:

- Decision: Special case? No. The body of `childMethod` does not fall into any special cases.
- Decision: In a function? Yes, we're inside of `childMethod`.
- Decision: Invoked as a method? Yes via dot notation (`Child.childMethod()`).
- Final Result: `this` equals left of dot, which is `Child`.

Special Case: `super.method()` (example 2 of 2)

This is an especially tricky example!

```
class Parent {
  static parentMethod() {
    console.log(this); // this?
  }
}

class Child extends Parent {
  static childMethod() {
```

```
    super.parentMethod();
  }
}

const test = Child.childMethod;
test();
```

What's gonna get logged here? The two classes are exactly the same as in the previous example. Only the last two lines are different. This example would be a tricky twist for most people, but I think it should be manageable for you if you methodically use the system. So before seeing my approach, try and see if you can get the answer on your own!

- Decision: Special case? Yes. We again fall into the `super.method()` special case.
- Intermediate Result: `this` equals outside `this` when function (`parentMethod`) is *invoked*.

Since `parentMethod` is invoked inside of `childMethod`, we must determine the `this` value inside of `childMethod`. To answer this question, we repeat the process:

- Decision: Special case? No. The body of `childMethod` does not into any special cases.
- Decision: In a function? Yes, we're inside of `childMethod`.
- Decision: Invoked as a method? No! (invoked via `test()`).
- Final Result: `this` equals `undefined` because `childMethod` *was* created on a class!

Bonus: Built-in functions

When you pass a function into a built-in function like `addEventListener` or `forEach`, you can't practically inspect the built-in function's source code. And that means you can't use our system to determine what `this` will be. So with that limitation, what can you do?

My favorite approach is to run experiments. I work to develop a hypothesis about what might be happening, and then carefully design experiments to test that hypothesis. I like this approach because you can generate extremely specific answers that can be hard to figure out any other way.

The most popular approach is to reach for documentation. This can be limiting though in cases where the documentation is incomplete or poorly written. Supplementing documentation with targeted experiments can be a powerful combination. This allows you to answer more specific questions and also turns reading into an active process of discovery.

The last resort is to read the underlying source code. In rare situations where you must know precisely what's happening, this may be unavoidable. Typical places to look for built-in JavaScript function are V8, Chromium, and Mozilla's SpiderMonkey.

The functions built-into JavaScript are sometimes implemented in JavaScript, but not always. For example, some are implemented in C++. So when you pass a function into a built-in function, even though the built-in function is *not* written in JavaScript, the built-in function will invoke *your* function in a way that's *functionally equivalent* to one of the flowchart cases.

That means you can safely operate as if all the built-in functions are written in JavaScript. So essentially, you can think of built-in functions as just normal JavaScript functions whose only shortcoming is that you can't easily see what's going on inside of them.

Built-in function: `addEventListener`

In this example, I want to run targeted experiments to understand how `addEventListener` works.

```
// Create a button, b.
const b = document.createElement('button');

// Add b to document.body.
document.body.appendChild(b);

// Add click handler.
b.addEventListener('click', function myCallback() {
  console.log(this); // this?
});
```

Each time we click the button `b`, the `this` value inside of our `myCallback` will get logged to the console.

The problem here is that some code kicked off by `addEventListener` will invoke `myCallback`, but we can't see how, because `addEventListener` is a built-in function.

So the next best thing is to see what it does and then try to reason about what might be happening. If you run the code and click `b`, we can see that `b` itself gets logged to the console. From here, we can use our flowchart to reverse-engineer what might be happening.

First, we consider whether we have a special case. We can't rule out `bind`, `call`, or `apply`. However, we can rule out `new` and `=>` because we're not in a constructor or arrow function. We can also rule out `super.method()` because we're not in a class.

Now let's consider whether a common case situation might apply. We're in a function, but we can't directly observe if it was invoked as a method. However, we can rule out the possibility that `myCallback` was *not* invoked as a method, because if it were, `this` would have to be either `window` or `undefined`.

Finally, `myCallback` could have been invoked in the common case, as a method. But it's very unlikely because `myCallback` would have to be a property of `b`. This would be fine if you have just one event listener, but an element can have many, so it could get really, really messy. So while theoretically possible, this scenario is extremely unlikely.

So that leaves just one plausible option, which is that `addEventListener` is explicitly setting the `this` value inside of `myCallback` to `b` using `bind`, `call`, or `apply`.

A common misconception about `addEventListener`

Many people use the mistake of assuming that `this` inside of the `myCallback` must be `b`, because `b` is to the left-of-the-dot in `b.addEventListener(function myCallback() {})`.

This *reasoning* is wrong! It is true, assuming no special cases, that `this` inside of `addEventListener` will be equal to `b`, because `addEventListener` is invoked as a method, but that says nothing about what `this` will be inside of `myCallback`, which is the function in question.

In order to think about `this` in `myCallback`, you must see how `myCallback` is invoked, and you simply cannot tell by looking at the code here. So be careful, it's really easy to make these subtle reasoning mistakes. In the course of doing research for this post, I saw several popular tutorial authors make this exact error!

It's disheartening to see so many "teachers" spreading this flawed thinking. Some might argue that it doesn't matter, because you can make this error and come to the right conclusion. But this argument essentially defends the broken clock that is right twice a day.

Built-in function: `forEach`

```
[1].forEach(function testFunction() {
  console.log(this);
});
```

In this example, I want to figure out how `forEach` works just by running experiments, without reading any documentation. `forEach` will run `testFunction`, but we can't see how, because `forEach` is a built-in function.

If we run the program, we see that `window` gets logged to the console. From here, we can use our flowchart to reverse-engineer what might be happening.

First, we consider whether we have a special case. We can't rule out `bind`, `call`, `apply` (Possibility A). However, we can rule out `new` and `=>` because we're not in a constructor or arrow function. We can also rule out `super.method()` because we're not in a class.

Now let's examine the common cases. We're in a function, so the next question in the flowchart is whether the function was invoked as a method.

If `testFunction` were *not* invoked as a method (Possibility B), `this` would be window (since the `testFunction` was not created on a class), which lines up with what's actually logged to the console.

If `testFunction` *were* invoked as a method (Possibility C), `testFunction` would have to temporarily be saved as a method on `window` at the time of invocation. I say temporarily, because if we inspect `window` after the program runs, we can see that no such method exists.

So that leaves us with 3 distinct possibilities that we must narrow down:

- Possibility A: `forEach` invokes `testFunction` with `bind`, `call`, or `apply` and explicitly sets `this` to `window`.
- Possibility B: `forEach` invokes `testFunction` in the common case, *not* as a method.
- Possibility C: `forEach` is invoking `testFunction` as a method on `window`.

After pondering this for a while with the flowchart in front of me, I wondered to myself, "what if we gave `forEach` a function that was initially created on a class?" Would that give us any useful information?

Stepping back, this is an application of a powerful problem-solving technique where you hold fixed a portion of the problem (in this case `forEach`) and vary the other elements (the function argument).

Here is the modified experiment:

```
class MyClass {
  static method() {
    console.log(this); // ?
  }
}

[1].forEach(MyClass.method);
```

From here, I thought about what would happen in each of the three possibilities (the "implications" I mentioned earlier):

- Possibility A: If this theory is true, `this` will still be `window`.
- Possibility B: If this theory is true, `this` will still be `undefined` since `method` was created on a class.
- Possibility C: If this theory is true, `this` will still be `window`.

So if we see `window` logged to the console, then it would mean that we either have Possibility A or Possibility C, and we'd have to do more work to narrow it down.

However, if we get `undefined`, then it would mean that it must be Possibility B. If we run the program, we see that `undefined` is logged to the console. That means we can know with certainty, that `forEach` invokes its argument in the common case, *not* as a method.

Thinking is the hard part

Armed with just a simple flowchart and basic problem-solving skills, you can answer extremely sophisticated questions that are impractical to figure out any other way.

Each time you see a piece of code, I want you to ask yourself, "what is `this` equal to"?" If you do that enough times, and introspect each time you're wrong, eventually you should be able to get it right every single time. If you do make a mistake, that's okay, but only if you take that as an opportunity to figure out what went wrong so that you can avoid making that mistake again.

That's the whole point of this type of deliberate practice. My main hope from writing this post has nothing to do with `this` keyword. I really couldn't care less. This kind of language specific thing to me at least, is the least interesting, least important, most boring part of programming.

My real hope is that you got a sense of how powerful it can be to approach problems in a more structured, more systematic way. All of my teaching is built on the similar idea that how you think is more important than what you know.

Anyone can memorize and regurgitate information, and computers are better than us at doing that anyway. Instead, I want you to get better at knowing what to do when you don't know the answer.

© 2025 Watch and Code