Introduction

The `this` keyword in JavaScript (and thus TypeScript) behaves differently than it does in many other languages. This can be very surprising, especially for users of other languages that have certain intuitions about how `this` should work.

This page will teach you how to recognize and diagnose problems with `this` in TypeScript, and describes several solutions and their respective trade-offs.

Typical Symptoms and Risk Factors

Typical symptoms of a lost `this` context include:

- A class field (`this.foo`) is `undefined` when some other value was expected
- The value `this` points to the global `window` object instead of the class instance (non-strict mode)
- The value `this` points `undefined` instead of the class instance (strict mode)
- Invoking a class method (`this.doBar()`) fails with the error "TypeError: undefined is not a function", "Object doesn't support property or method 'doBar'", or "this.doBar is not a function"

These things often happen in certain coding patterns:

- Event listeners, e.g. `window.addEventListener('click', myClass.doThing);`
- Promise resolution, e.g. `myPromise.then(myClass.theNextThing);`
- Library event callbacks, e.g. `$(document).ready(myClass.start);`
- Functional callbacks, e.g. `someArray.map(myClass.convert)`
- Classes in ViewModel-type libraries, e.g. `<div data-bind="click: myClass.doSomething">`
- Functions in options bags, e.g. `$.ajax(url, { success: myClass.handleData })`

What is `this` in JavaScript?

Much has been written about the hazards of `this` in JavaScript. See this page, this one, or this other one.

When a function is invoked in JavaScript, you can follow these steps to determine what `this` will be (these rules are in priority order):

- If the function was the result of a call to `function#bind` , `this` will be the argument given to `bind`
- If the function was invoked in the form `foo.func()` , `this` will be `foo`
- If in strict mode, `this` will be `undefined`
- Otherwise, `this` will be the global object (`window` in a browser)

These rules can result in some counter-intuitive behavior. For example:

```
class Foo {
  x = 3;
  print() {
    console.log('x is ' + this.x);
  }
}

var f = new Foo();
f.print(); // Prints 'x is 3' as expected
```

```
// Use the class method in an object literal
var z = { x: 10, p: f.print };
z.p(); // Prints 'x is 10'

var p = z.p;
p(); // Prints 'x is undefined'
```

Red Flags for `this`

The biggest red flag you can keep in mind is *the use of a class method without immediately invoking it*. Any time you see a class method being *referenced* without being *invoked* as part of that same expression, `this` might be incorrect.

Examples:

```
var x = new MyObject();
x.printThing(); // SAFE, method is invoked where it is referenced

var y = x.printThing; // DANGER, invoking 'y()' may not have correct 'this'

window.addEventListener('click', x.printThing, 10); // DANGER, method is not invoked
where it is referenced

window.addEventListener('click', () => x.printThing(), 10); // SAFE, method is
invoked in the same expression
```

Fixes

There are several ways to correctly keep your `this` context.

Use Instance Functions

Instead of using a *prototype* method, the default for methods in TypeScript, you can use an *instance arrow function* to define a class member:

```
class MyClass {
    private status = "blah";

    public run = () => { // <-- note syntax here
        alert(this.status);
    }
}
var x = new MyClass();
$(document).ready(x.run); // SAFE, 'run' will always have correct 'this'
```

- Good/bad: This creates an additional closure per method per instance of the class. If this method is usually only used in regular method calls, this is overkill. However, if it's used a lot in callback positions, it's more efficient for the class instance to capture the `this` context instead of each call site creating a new closure upon invoke.
- Good: Impossible for external callers to forget to handle `this` context

- Good: Typesafe in TypeScript
- Good: No extra work if the function has parameters
- Bad: Derived classes can't call base class methods written this way using `super`
- Bad: The exact semantics of which methods are "pre-bound" and which aren't create an additional non-typesafe contract between the class and its consumers

Local Fat Arrow

In TypeScript (shown here with some dummy parameters for explanatory reasons):

```
var x = new SomeClass();
someCallback((n, m) => x.doSomething(n, m));
```

- Good/bad: Opposite memory/performance trade-off compared to instance functions
- Good: In TypeScript, this has 100% type safety
- Good: Works in ECMAScript 3
- Good: You only have to type the instance name once
- Bad: You'll have to type the parameters twice
- Bad: Doesn't work with variadic ('rest') parameters

Function.bind

```
var x = new SomeClass();
// SAFE: Functions created from function.bind always preserve 'this'
window.setTimeout(x.someMethod.bind(x), 100);
```

- Good/bad: Opposite memory/performance trade-off compared to using instance functions
- Good: No extra work if the function has parameters
- Bad: In TypeScript, this currently has no type safety
- Bad: Only available in ECMAScript 5 or newer
- Bad: You have to type the instance name twice

Specify type of `this` in function signature

See details here.

```
interface SomeEvent {
    cancelable: boolean;
    preventDefault(): void;
}

function eventHandler(this: SomeEvent) {
    if (this.cancelable) {
        this.preventDefault();
    }
    // ...
}
```

- Good: The function has type information of the context it is supposed to run in, which is helpful in type checking and IDE completion

- Bad: The syntax of having `this` type declaration among function arguments might be confusing for developers at reading-time