This

Why `this` ? **`This`** provides a way of implicitly passing along an object reference.

2 misconceptions about `this` :

- this refers to the function itself
- this refers to the function scope

```
//misconception #1 - function itself
function foo(num) {
  console.log('foo' + num);
  this.count++;
}
foo.count = 0;
foo(1);
foo(2);
console.log(foo.count); // 0 not 2

//misconception #2 - function scope
function foo() {
  var a = 2;
  this.bar()
}

function bar() {
  console.log(this.a)
}

foo() //undefined
```

The mechanics behind those examples will be explained below - but they prove the misconception.

So what is really `this` ?

`This` binding has nothing to do with where a function is declared, but has instead everything to do with the manner in which the function is called

Call Site

The Call Site is the location in the code where a function is called.

As a general rule the call stack provides you the call site. The call site of a function you're breaked on, will be located in the previous line of the call stak you're inspecting.

```
function bar() {
  // call stack is : bar
  // call site is in global scope
  foo() //call site for foo
}
function foo() {
  // call stack is : bar -> foo
  // therefore call site is in bar
  ...
}
bar()
```

Based on the call site there are 4 rules that drives the `this` binding.

They are applied in this specific order, meaning the first one takes precendence on the others

1. **Default** Binding
2. **Implicit** Binding
3. **Explicit** Binding
4. **New** Binding

Default Binding

This is the most common case : standalone function invocation.

When there is nothing preceding the function call - `this` value will be the **global object**

```
var a = 2
function foo(){
  console.log(this.a);
}

foo() // 2
```

using let or const do not add the variables as properties on the global object

```
const a = 2
function foo(){
console.log(this.a);
}
foo() // undefined
// this is still defined as the global object, but there is no a property on it
```

In strict mode the global object is not eligible

```
function foo(){
"use strict"
  console.log(this.a);
}
const a = 2;
foo() // TypeError: this is undefined
```

Implicit Binding

If the call site has a context object. This litteraly means that the function call is preceded by an object reference.

When there is a context object, `this` value will be the **context object itself**

```
function foo(){
  console.log(this.a)
}

const obj = {
  a:2,
  foo: foo
}
obj.foo() // 2
```

One of the most common pitfall with implicit binding is when we loose context

```
function foo() {
  console.log(this.a)
}

const obj = {
  a:2,
  foo : foo
}
var a = 'ooops global'
var bar = obj.foo; //function reference alias

bar() // ooops global
```

`bar` seems to be a reference to `obj.foo` but in fact is really just an other reference to `foo` itself. **The call site** is what matters and the call site is a global non decorated `bar()`.

The same happens with callback :

```
function foo() {
    console.log(this.a)
}

const obj = {
  a:2,
  foo:foo
}

setTimeout(obj.foo, 100) // undefined
```

Parameter passing is just an implicit assignment so the end result is the same as the previous snippet.

Explicit Binding

What if you want to force a function call to use a particular object for `this` binding without putting a method on the object ?

There exists 3 different built in method for this purpose :

- `call`, `apply` and `bind`

```
function foo() {
  console.log(this.a)
}

var obj = {
  a:2
}
foo.call(obj) //2
foo.apply(obj) //2
```

```
function foo() {
    console.log(this.a)
}

const obj = {
  a:2,
  foo:foo
}

setTimeout(foo.apply(obj), 100) // 2
```

call and **apply** both force the function call `this` to the object passed. Both can take additional arguments to be passed to the initial function.

In `call` the subsequent arguments are passed in to the function as they are, while `apply` expects the second argument to be an array that it unpacks as arguments for the called function.

```
function foo(something) {
  console.log(this.a, something)
  return this.a + something
}

var obj = {
  a:2
}

var bar = foo.bind(obj)

var b = bar(3)
console.log(b) //5
```

ES5 introduced **bind**, a utility wrapper, which returns a new function that is hardcoded to call the orifinal function with the `this` context set as you specified

New binding

When a function is invoked with the `new` keyword in front of it, the following things are done aurtomatically:

- a brand new object is created
- the newly constructed object is `[[Prototype]]`-linked.
- the newly constructed object is set as the `this` binding for that function call
- unless the function returns its own alternate object, the newly function call will automatically return the new object created

The 4 rules are applied in this order (if the first one isn't valid, the second one is considered). If function is :

- called with `new` binding. If so `this` is the **newly constructed object**
- called with `call` or `apply`. If so `this` is **the explicitely specified object**
- called with a context object. If so `this` is **the context object**
- otherwise `this` is **global object** or **undefined** depending on strict mode

Arrow function

The arrow function `() => …` does not use the same rules. Instead it adopts the `this` binding from the enclosing scope.(function or global)

```
function foo() {
  setTimeout(() => console.log(this.a), 1000)
}
const obj = {
  a: 2
}
foo.call(obj) //2 - the this value is not lost with the parameter assignment
```