this

A function's `this` keyword behaves a little differently in JavaScript compared to other languages. It also has some differences between strict mode and non-strict mode.

In most cases, the value of `this` is determined by how a function is called (runtime binding). It can't be set by assignment during execution, and it may be different each time the function is called. The `bind()` method can set the value of a function's `this` regardless of how it's called, and arrow functions don't provide their own `this` binding (it retains the `this` value of the enclosing lexical context).

```
const test = {
  prop: 42,
  func: function() {
    return this.prop;
  },
};

console.log(test.func());
// Expected output: 42
```

Syntax

```
this
```

Value

In non–strict mode, `this` is always a reference to an object. In strict mode, it can be any value. For more information on how the value is determined, see the description below.

Description

The value of this depends on in which context it appears: function, class, or global.

Function context

When you write a function in your code, it's like a little machine that does some work for you. The function might need to know some information to do its job, so you can give it some parameters to use.

But there's one more special thing that the function has: it has a secret code word called "`this`". The value of "`this`" depends on how you use the function. If you use the function with an object like this: `obj.f()`, then "`this`" means `obj`.

Think of "`this`" like a hidden extra parameter that the function gets without you having to write it down. It helps the function know what object it should be working with.

For example:

```
function getThis() {
  return this;
}

const obj1 = { name: "obj1" };
const obj2 = { name: "obj2" };

obj1.getThis = getThis;
obj2.getThis = getThis;

console.log(obj1.getThis()); // { name: 'obj1', getThis: [Function: getThis] }
console.log(obj2.getThis()); // { name: 'obj2', getThis: [Function: getThis] }
```

Note how the function is the same, but based on how it's invoked, the value of this is different. This is analogous to how function parameters work.

The value of `this` is not the object that has the function as an own property, but the object that is used to call the function. You can prove this by calling a method of an object up in the prototype chain.

```
const obj3 = {
  __proto__: obj1,
  name: "obj3",
};
console.log(obj3.getThis()); // { name: 'obj3' }
```

The value of `this` always changes based on how a function is called, even when the function was defined on an object at creation:

```
const obj4 = {
  name: "obj4",
  getThis() {
    return this;
  },
};

const obj5 = { name: "obj5" };

obj5.getThis = obj4.getThis;
console.log(obj5.getThis()); // { name: 'obj5', getThis: [Function: getThis] }
```

If the value that the method is accessed on is a primitive, `this` will be a primitive value as well — but only if the function is in strict mode.

```
function getThisStrict() {
  "use strict"; // Enter strict mode
  return this;
}

// Only for demonstration — you should not mutate built-in prototypes
Number.prototype.getThisStrict = getThisStrict;
console.log(typeof (1).getThisStrict()); // "number"
```

If the function is called without being accessed on anything, `this` will be `undefined` — but only if the function is in strict mode.

```
console.log(typeof getThisStrict()); // "undefined"
```

In non-strict mode, a special process called this substitution ensures that the value of this is always an object. This means:

- If a function is called with `this` set to undefined or `null`, this gets substituted with `globalThis`.
- If the function is called with `this` set to a primitive value, this gets substituted with the primitive value's wrapper object.

```
function getThis() {
  return this;
}

// Only for demonstration — you should not mutate built-in prototypes
Number.prototype.getThis = getThis;
console.log(typeof (1).getThis()); // "object"
console.log(getThis() === globalThis); // true
```

In typical function calls, this is implicitly passed like a parameter through the function's prefix (the part before the dot). You can also explicitly set the value of this using the `Function.prototype.call()`, `Function.prototype.apply()`, or `Reflect.apply()` methods. Using `Function.prototype.bind()`, you can create a new function with a specific value of this that doesn't change regardless of how the function is called. When using these methods, the this substitution rules above still apply if the function is non-strict.

Callbacks

When a function is passed as a callback, the value of `this` depends on how the callback is called, which is determined by the implementor of the API. Callbacks are typically called with a `this` value of `undefined` (calling it directly without attaching it to any object), which means if the function is non–strict, the value of this is the global object (`globalThis`). This is the case for iterative array methods, the `Promise()` constructor, etc.

```
function logThis() {
  "use strict";
  console.log(this);
}

[1, 2, 3].forEach(logThis); // undefined, undefined, undefined
```

Some APIs allow you to set a `this` value for invocations of the callback. For example, all iterative array methods and related ones like `Set.prototype.forEach()` accept an optional `thisArg` parameter.

```
[1, 2, 3].forEach(logThis, { name: "obj" });
// { name: 'obj' }, { name: 'obj' }, { name: 'obj' }
```

Occasionally, a callback is called with a `this` value other than undefined. For example, the reviver parameter of `JSON.parse()` and the `replacer` parameter of `JSON.stringify()` are both called with `this` set to the object that the property being parsed/serialized belongs to.

Arrow functions

In arrow functions, `this` retains the value of the enclosing lexical context's `this`. In other words, when evaluating an arrow function's body, the language does not create a new `this` binding.

For example, in global code, `this` is always `globalThis` regardless of strictness, because of the global context binding:

```
const globalObject = this;
const foo = () => this;
console.log(foo() === globalObject); // true
```

Arrow functions create a closure over the `this` value of its surrounding scope, which means arrow functions behave as if they are "auto-bound" — no matter how it's invoked, `this` is set to what it was when the function was created (in the example above, the global object). The same applies to arrow functions created inside other functions: their `this` remains that of the enclosing lexical context. See example below.

Furthermore, when invoking arrow functions using `call()`, `bind()`, or `apply()`, the `thisArg` parameter is ignored. You can still pass other arguments using these methods, though.

```
const obj = { name: "obj" };

// Attempt to set this using call
console.log(foo.call(obj) === globalObject); // true

// Attempt to set this using bind
const boundFoo = foo.bind(obj);
console.log(boundFoo() === globalObject); // true
```

Constructors

When a function is used as a constructor (with the `new` keyword), its `this` is bound to the new object being constructed, no matter which object the constructor function is accessed on. The value of `this` becomes the value of the `new` expression unless the constructor returns another non–primitive value.

```
function C() {
  this.a = 37;
}

let o = new C();
console.log(o.a); // 37

function C2() {
  this.a = 37;
  return { a: 38 };
}

o = new C2();
console.log(o.a); // 38
```

In the second example (`C2`), because an object was returned during construction, the new object that this was bound to gets discarded. (This essentially makes the statement `this.a = 37;` dead code. It's not exactly dead because it gets executed, but it can be eliminated with no outside effects.)

super

When a function is invoked in the `super.method()` form, the `this` inside the `method` function is the same value as the `this` value around the `super.method()` call, and is generally not equal to the object that `super` refers to. This is because super.method is not an object member access like the ones above — it's a special syntax with different binding rules. For examples, see the `super` reference.

Class context

A class can be split into two contexts: static and instance. Constructors, methods, and instance field initializers (public or private) belong to the instance context. Static methods, static field initializers, and static initialization blocks belong to the static context. The `this` value is different in each context.

Class constructors are always called with `new`, so their behavior is the same as function constructors: the `this` value is the new instance being created. Class methods behave like methods in object literals — the `this` value is the object that the method was accessed on. If the method is not transferred to another object, `this` is generally an instance of the class.

Static methods are not properties of `this`. They are properties of the class itself. Therefore, they are generally accessed on the class, and `this` is the value of the class (or a subclass). Static initialization blocks are also evaluated with `this` set to the current class.

Field initializers are also evaluated in the context of the class. Instance fields are evaluated with `this` set to the instance being constructed. Static fields are evaluated with `this` set to the current class. This is why arrow functions in field initializers are bound to the class.

```
class C {
  instanceField = this;
  static staticField = this;
}

const c = new C();
console.log(c.instanceField === c); // true
console.log(C.staticField === C); // true
```

Derived class constructors

Unlike base class constructors, derived constructors have no initial `this` binding. Calling `super()` creates a `this` binding within the constructor and essentially has the effect of evaluating the following line of code, where `Base` is the base class:

```
this = new Base();
```

Warning: Referring to `this` before calling `super()` will throw an error.

Derived classes must not return before calling `super()`, unless the constructor returns an object (so the `this` value is overridden) or the class has no constructor at all.

```
class Base {}
class Good extends Base {}
class AlsoGood extends Base {
  constructor() {
    return { a: 5 };
  }
}
```

```
class Bad extends Base {
  constructor() {}
}

new Good();
new AlsoGood();
new Bad(); // ReferenceError: Must call super constructor in derived class before
              accessing 'this' or returning from derived constructor
```

Global context

In the global execution context (outside of any functions or classes; may be inside blocks or arrow functions defined in the global scope), the `this` value depends on what execution context the script runs in. Like callbacks, the `this` value is determined by the runtime environment (the caller).

At the top level of a script, `this` refers to `globalThis` whether in strict mode or not. This is generally the same as the global object — for example, if the source is put inside an HTML `<script>` element and executed as a script, `this === window`.

Note: `globalThis` is generally the same concept as the global object (i.e. adding properties to `globalThis` makes them global variables) — this is the case for browsers and Node — but hosts are allowed to provide a different value for `globalThis` that's unrelated to the global object.

```
// In web browsers, the window object is also the global object:
console.log(this === window); // true

this.b = "MDN";
console.log(window.b); // "MDN"
console.log(b); // "MDN"
```

If the source is loaded as a module (for HTML, this means adding `type="module"` to the `<script>` tag), `this` is always `undefined` at the top level.

If the source is executed with `eval()`, this is the same as the enclosing context for direct eval, or `globalThis` (as if it's run in a separate global script) for indirect eval.

```
function test() {
  // Direct eval
  console.log(eval("this") === this);
  // Indirect eval, non-strict
  console.log(eval?.("this") === globalThis);
  // Indirect eval, strict
  console.log(eval?.("'use strict'; this") === globalThis);
}

test.call({ name: "obj" }); // Logs 3 "true"
```

Note that some source code, while looking like the global scope, is actually wrapped in a function when executed. For example, Node.js CommonJS modules are wrapped in a function and executed with the `this` value set to `module.exports`. Event handler attributes are executed with `this` set to the element they are attached to.

Object literals don't create a `this` scope — only functions (methods) defined within the object do. Using `this` in an object literal inherits the value from the surrounding scope.

```
const obj = {
  a: this,
};

console.log(obj.a === window); // true
```

Examples

this in function contexts

The value of this depends on how the function is called, not how it's defined.

```
// An object can be passed as the first argument to call
// or apply and this will be bound to it.
const obj = { a: "Custom" };

// Variables declared with var become properties of the global object.
var a = "Global";

function whatsThis() {
  return this.a; // The value of this is dependent on how the function is called
}
```

```
whatsThis(); // 'Global'; this in the function isn't set, so it defaults to the
                 global/window object in non–strict mode
obj.whatsThis = whatsThis;
obj.whatsThis(); // 'Custom'; this in the function is set to obj
```

Using `call()` and `apply()`, you can pass the value of this as if it's an actual parameter.

this and object conversion

In non–strict mode, if a function is called with a `this` value that's not an object, the this value is substituted with an object. `null` and `undefined` become `globalThis`. Primitives like `7` or `'foo'` are converted to an object using the related constructor, so the primitive number `7` is converted to a `Number` wrapper class and the string `'foo'` to a `String` wrapper class.

```
function bar() {
  console.log(Object.prototype.toString.call(this));
}

bar.call(7); // [object Number]
bar.call("foo"); // [object String]
bar.call(undefined); // [object Window]
```

The bind() method

Calling `f.bind(someObject)` creates a new function with the same body and scope as `f`, but the value of `this` is permanently bound to the first argument of `bind`, regardless of how the function is being called.

```
function f() {
  return this.a;
}

const g = f.bind({ a: "azerty" });
console.log(g()); // azerty

const h = g.bind({ a: "yoo" }); // bind only works once!
console.log(h()); // azerty

const o = { a: 37, f, g, h };
console.log(o.a, o.f(), o.g(), o.h()); // 37,37, azerty, azerty
```

this in arrow functions

Arrow functions create closures over the `this` value of the enclosing execution context. In the following example, we create `obj` with a method `getThisGetter` that returns a function that returns the value of `this`. The returned function is created as an arrow function, so its `this` is permanently bound to the `this` of its enclosing function. The value of this inside `getThisGetter` can be set in the call, which in turn sets the return value of the returned function.

```
const obj = {
  getThisGetter() {
    const getter = () => this;
    return getter;
  },
};
```

We can call `getThisGetter` as a method of `obj`, which sets this inside the body to `obj`. The returned function is assigned to a variable `fn`. Now, when calling `fn`, the value of this returned is still the one set by the call to `getThisGetter`, which is obj. If the returned function is not an arrow function, such calls would cause the `this` value to be `globalThis` or undefined in strict mode.

```
const fn = obj.getThisGetter();
console.log(fn() === obj); // true
```

But be careful if you unbind the method of `obj` without calling it, because `getThisGetter` is still a method that has a varying this value. Calling `fn2()()` in the following example returns `globalThis`, because it follows the this from `fn2`, which is `globalThis` since it's called without being attached to any object.

```
const fn2 = obj.getThisGetter;
console.log(fn2()() === globalThis); // true
```

This behavior is very useful when defining callbacks. Usually, each function expression creates its own `this` binding, which shadows the `this` value of the upper scope. Now, you can define functions as arrow functions if you don't care about the `this` value, and only create this bindings where you do (e.g. in class methods). See example with `setTimeout()`.

this with a getter or setter

`this` in getters and setters is based on which object the property is accessed on, not which object the property is defined on. A function used as getter or setter has its `this` bound to the object from which the property is being set or gotten.

```
function sum() {
  return this.a + this.b + this.c;
}

 const o = {
  a: 1,
  b: 2,
  c: 3,
  get average() {
    return (this.a + this.b + this.c) / 3;
  },
};

Object.defineProperty(o, "sum", {
  get: sum,
  enumerable: true,
  configurable: true,
});

console.log(o.average, o.sum); // 2, 6
```

As a DOM event handler

When a function is used as an event handler, its `this` is set to the element on which the listener is placed (some browsers do not follow this convention for listeners added dynamically with methods other than `addEventListener()`).

```
// When called as a listener, turns the related element blue
function bluify(e) {
  // Always true
  console.log(this === e.currentTarget);
  // true when currentTarget and target are the same object
  console.log(this === e.target);
  this.style.backgroundColor = "#A5D9F3";
}
```

```
// Get a list of every element in the document
const elements = document.getElementsByTagName("*");

// Add bluify as a click listener so when the
// element is clicked on, it turns blue
for (const element of elements) {
  element.addEventListener("click", bluify, false);
}
```

this in inline event handlers

When the code is called from an inline event handler attribute, its `this` is set to the DOM element on which the listener is placed:

```
<button onclick="alert(this.tagName.toLowerCase());">Show this</button>
```

The above alert shows button. Note, however, that only the outer code has its `this` set this way:

```
<button onclick="alert((function () { return this; })());">
  Show inner this
</button>
```

In this case, the inner function's `this` isn't set, so it returns the global/window object (i.e. the default object in non–strict mode where `this` isn't set by the call).

Bound methods in classes

Just like with regular functions, the value of `this` within methods depends on how they are called. Sometimes it is useful to override this behavior so that `this` within classes always refers to the class instance. To achieve this, bind the class methods in the constructor:

```
class Car {
  constructor() {
    // Bind sayBye but not sayHi to show the difference
    this.sayBye = this.sayBye.bind(this);
  }
sayHi() {
    console.log(`Hello from ${this.name}`);
  }
  sayBye() {
    console.log(`Bye from ${this.name}`);
  }
  get name() {
    return "Ferrari";
  }
}

class Bird {
  get name() {
    return "Tweety";
  }
}

const car = new Car();
const bird = new Bird();

// The value of 'this' in methods depends on their caller
car.sayHi(); // Hello from Ferrari
bird.sayHi = car.sayHi;
bird.sayHi(); // Hello from Tweety

// For bound methods, 'this' doesn't depend on the caller
bird.sayBye = car.sayBye;
bird.sayBye(); // Bye from Ferrari
```

Note: Classes are always in strict mode. Calling methods with an undefined `this` will throw an error if the method tries to access properties on `this`.

Note, however, that auto-bound methods suffer from the same problem as using arrow functions for class properties: each instance of the class will have its own copy of the method, which increases memory usage. Only use it where absolutely necessary. You can also mimic the implementation of `Intl.NumberFormat.prototype.format()`: define the property as a getter that returns a bound function when accessed and saves it, so that the function is only created once and only created when necessary.

this in with statements

Although with statements are deprecated and not available in strict mode, they still serve as an exception to the normal this binding rules. If a function is called within a with statement and that function is a property of the scope object, the `this` value is set to the scope object, as if the `obj1.` prefix exists.

```
const obj1 = {
  foo() {
    return this;
  },
};

with (obj1) {
  console.log(foo() === obj1); // true
}
```