The this keyword

The keyword `this` refers to the value of the object that is bound to the function at the time of its call, meaning that its value is different depending on whether a function is called as a method, as a standalone function, or as a constructor (/learn/javascript/functions/new).

When a function is called, it creates an instance of the keyword `this` behind the scenes as a reference to the object that contains that function, giving access to the properties and methods defined alongside it from within its scope. Working with `this` is similar in some ways to working with a variable declared with `const`. Like a constant, `this` can't be removed and its value can't be reassigned, but the methods and properties of the object that the `this` keyword contains can be altered.

Global binding

Outside a function or the context of an object, `this` refers to the `globalThis` property, which is a reference to the global object in most JavaScript environments. In the context of a script running in a web browser, the global object is the `window` object:

```
this;
> Window {0: Window, window: Window, self: Window, document: document, name: '', l
```

In Node.js, `globalThis` is the `global` object:

```
$ node
Welcome to Node.js v20.10.0.
Type ".help" for more information.
> this
<ref *1> Object [global] {
```

```
...
}
```

Outside strict mode, `this` also refers to the global object inside a standalone function, because the parent `Window` is the object that effectively "owns" those functions.

```
function myFunction() {
    console.log( this );
}
myFunction();
> Window {...}

(function() {
    console.log( this );
}());
> Window {...}
```

When using strict mode, `this` has a value of `undefined` inside a standalone function:

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

Before the introduction of strict mode, a `null` or `undefined` value for `this` would be replaced by a reference to the global object. You might sometimes see global binding referred to as "default binding" because of this legacy behavior.

Implicit binding

When a function is called as a method of an object, an instance of `this` inside that method refers to the object that contains the method, giving access to the methods and properties that sit alongside it:

```
let myObject = {
    myValue: "This is my string.",
    myMethod() {
            console.log( this.myValue );
    }
};

myObject.myMethod();
> "This is my string."
```

It might look like the value of `this` depends on how a function and its enclosing object are defined. Instead, the context for the value of `this` is the current *execution* context. In this case, the execution context is that the `myObject` object is calling the `myMethod` method, so `myObject` is the value for `this`. This might seem like a technicality in the context of the previous examples, but for more advanced uses of `this`, it's an essential distinction to keep in mind.

In general, use `this` in ways that don't expect the surrounding code to have any particular structure. The exception to this rule is ES5 Arrow functions (/learn/javascript/functions/function-expressions#arrow-functions).

`this` in arrow functions

In arrow functions (/learn/javascript/functions/function-expressions#arrow-functions), `this` resolves to a binding in a *lexically enclosing environment* (https://262.ecma-international.org/6.0/#sec-arrow-function-definitions-runtime-semantics-evaluation). This means that `this` in an arrow function refers to the value of `this` in that function's closest enclosing context:

```
let myObject = {
    myMethod() { console.log( this ); },
    myArrowFunction: () => console.log( this ),
    myEnclosingMethod: function () {
        this.myArrowFunction = () => { console.log(this) };
    }
};

myObject.myMethod();
> Object { myMethod: myMethod(), myArrowFunction: myArrowFunction() }

myObject.myArrowFunction();
```

```
> Window {...}
```

In the previous example, `myObject.myMethod()` logs `myObject` as the object that "owns" that method, but `myObject.myArrowFunction()` returns `globalThis` (or `undefined`), because the instance of `this` inside the arrow function refers instead to the highest enclosing scope.

In the following example, `myEnclosingMethod` creates an arrow function on the object that contains it when it's executed. The instance of `this` inside the arrow function now refers to the value of `this` inside the enclosing environment, which is the method that contains that arrow function. Because the value of `this` inside `myEnclosingMethod` refers to `myObject`, after you define the arrow function, `this` inside the arrow function also refers to `myObject`:

```
let myObject = {
    myMethod() { console.log( this ); },
    myEnclosingMethod: function () {
        this.myArrowFunction = () => { console.log(this) };
    }
};

myObject.myEnclosingMethod();
myObject.myArrowFunction();
> Object { myMethod: myMethod(), myArrowFunction: myArrowFunction() }
```

Explicit binding

Implicit binding handles most use cases for working with `this`. However, you might sometimes need the value of `this` to represent a *specific* execution context, instead of the assumed context. An illustrative, if slightly outdated, example is working with `this` within the callback function of a `setTimeout`, because this callback has a unique execution context:

```
var myObject = {
  myString: "This is my string.",
  myMethod() {
    console.log( this.myString );
  }
};
myObject.myMethod();
> "This is my string."
```

```
setTimeout( myObject.myMethod, 100 );
> undefined
```

Although this specific shortcoming of `setTimeout` has since been addressed by other features, similar issues of "losing" `this` have previously been addressed by creating an explicit reference to the value of `this` within the scope of the intended context. You might occasionally see instances of `this` being assigned to a variable using identifiers like `that`, `self`, or `_this` in legacy codebases. These are common identifier conventions for variables containing a passed `this` value.

When you call a function using the `call()`, `bind()`, or `apply()` methods, `this` explicitly references the object being called:

```
let myFunction = function() {
    console.log( this.myValue );
}

let myObject = {
   "myValue" : "This is my string."
 };

myFunction.call( myObject );
> "This is my string."
```

```
var myObject = {
  myString: "This is my string.",
  myMethod() {
    console.log( this.myString );
  }
};

setTimeout( myObject.myMethod.bind( myObject ), 100 );
> "This is my string."
```

Explicit binding overrides the `this` value provided by implicit binding.

```
let myObject = {
    "myValue" : "This string sits alongside myMethod.",
    myMethod() {
        console.log( this.myValue );
    }
};
let myOtherObject = {
    "myValue" : "This is a string in another object entirely.",
};

myObject.myMethod.call( myOtherObject );
> "This is a string in another object entirely."
```

If a function is called in a way that would set the value of `this` to `undefined` or `null`, that value is replaced by `globalThis` outside strict mode:

```
let myFunction = function() {
    console.log( this );
}

myFunction.call( null );
> Window {...}
```

Similarly, if a function is called in a way that would give `this` a primitive value, that value is substituted with the primitive value's wrapper object (/learn/javascript/appendix#prototypal-inheritance) outside strict mode:

```
let myFunction = function() {
    console.log( this );
}

let myNumber = 10;

myFunction.call( myNumber );
> Number { 10 }
```

In strict mode, a passed `this` value isn't coerced to an object in any way, even if it's a primitive, `null`, or `undefined` value:

```
"use strict";
let myFunction = function() {
    console.log( this );
}

let myNumber = 10;

myFunction.call( myNumber );
> 10

myFunction.call( null );
> null
```

new binding

When a class (/learn/javascript/classes) is used as a constructor using the `new` keyword, `this` refers to the newly-created instance:

```
class MyClass {
    myString;
    constructor() {
        this.myString = "My string.";
    }
    logThis() {
        console.log( this );
    }
}
const thisClass = new MyClass();

thisClass.logThis();
> Object { myString: "My string." }
```

Similarly, the value of `this` inside a constructor function called using `new` refers to the object being created:

```
function MyFunction() {
  this.myString = "My string.";
  this.logThis = function() {
    console.log( this );
  }
```

```
}
const myObject = new MyFunction();

myObject.logThis();
> Object { myString: "My string.", logThis: logThis() }
```

Event handler binding

In the context of event handlers, the value of `this` references the object that invokes it. Inside an event handler's callback function, that means `this` references the element associated with the handler:

```
let button = document.querySelector( "button" );

button.addEventListener( "click", function( event ) { console.log( this ); } );
```

When a user interacts with the `button` in the previous snippet, the result is the element object containing the `<button>` itself:

```
> Button {}
```

When an arrow function is used as an event listener callback, the value of `this` is again provided by the closest enclosing execution context. At the top level, that means `this` inside an event handler callback function is `globalThis`:

```
let button = document.querySelector( "button" );

button.addEventListener( "click", ( event ) => { console.log( this ); } );
> undefined
```

As with any other object, when you use the `call()`, `bind()`, or `apply()` methods to reference the callback function of an event listener, `this` references the object explicitly:

```
let button = document.querySelector( "button" );
let myObject = {
    "myValue" : true
};
function handleClick() {
    console.log( this );
}

button.addEventListener( "click", handleClick.bind( myObject ) );
> Object { myValue: true }
```

Check your understanding

For a script running in a web browser, what is the global object that `this` refers to when used outside a function or the context of an object?

- [ ] The `undefined` object
- [ ] The `window` object
- [ ] The `browser` object

Last updated 2024-03-31 UTC.