This Keyword

The this keyword is one of the most widely used and yet confusing keyword in JavaScript. Here, you will learn everything about this keyword.

this points to a particular object. Now, which is that object is depends on how a function which includes 'this' keyword is being called.

```
<script>
var myVar = 100;

function WhoIsThis() {
    var myVar = 200;

    alert(myVar); // 200
    alert(this.myVar); // 100
}

WhoIsThis(); // inferred as window.WhoIsThis()

var obj = new WhoIsThis();
alert(obj.myVar);
</script>
```

The following four rules applies to this in order to know which object is referred by this keyword.

- **Global Scope**
- **Object's Method**
- **call() or apply() method**
- **bind() method**

Global Scope

If a function which includes 'this' keyword, is called from the global scope then this will point to the window object.

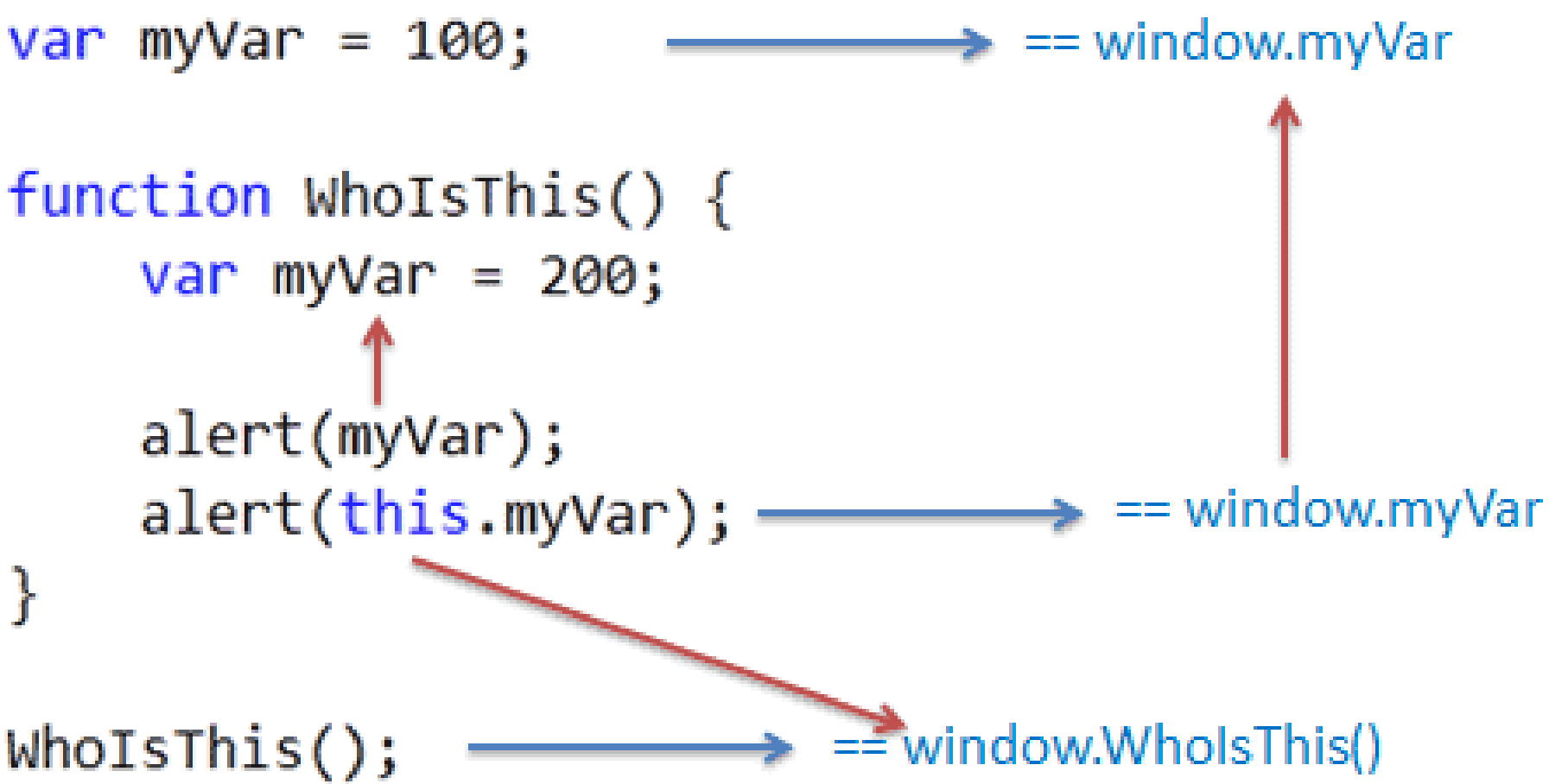
```
<script>
var myVar = 100;

function WhoIsThis() {
    var myVar = 200;

    alert("myVar = " + myVar); // 200
    alert("this.myVar = " + this.myVar); // 100
}

WhoIsThis(); // inferred as window.WhoIsThis()
</script>
```

In the example, a function WhoIsThis() is being called from the global scope. The global scope means in the context of window object. We can optionally call it like window.WhoIsThis(). So in the above example, this keyword in WhoIsThis() function will refer to window object. So, this.myVar will return 100. However, if you access myVar without this then it will refer to local myVar variable defined in WhoIsThis() function.


var myVar = 100;
== window.myVar
function WhoIsThis() {
var myVar = 200;
alert(myVar);
alert(this.myVar);
== window.myVar
}
WhoIsThis();
== window.WhoIsThis()
© TutorialsTeacher.com

'this' points to global window object even if it is used in an inner function. Consider the following example.

```
var myVar = 100;

function SomeFunction() {

    function WhoIsThis() {
        var myVar = 200;

        alert("myVar = " + myVar); // 200
        alert("this.myVar = " + this.myVar); // 100
    }

    WhoIsThis();
}

SomeFunction();
```

So, if 'this' is used inside any global function and called without dot notation or using window. then this will refer to global object which is default window object.

“this” Inside Object's Method

We can create an object of a function using new keyword. So, when you create an object of a function using new keyword then this will point to that particular object.

```
var myVar = 100;

function WhoIsThis() {
   this.myVar = 200;
}
var obj1 = new WhoIsThis();

var obj2 = new WhoIsThis();
obj2.myVar = 300;

alert(obj1.myVar); // 200
alert(obj2.myVar); // 300
```

In the above example, this points to obj1 for obj1 instance and points to obj2 for obj2 instance. In JavaScript, properties can be attached to an object dynamically using dot notation. Thus, myVar will be a property of both the instances and each will have a separate copy of myVar.

```
var myVar = 100;

function WhoIsThis() {
   this.myVar = 200;

   this.display = function(){
      var myVar = 300;

      alert("myVar = " + myVar); // 300
      alert("this.myVar = " + this.myVar);  // 200
   };
}
var obj = new WhoIsThis();

obj.display();
```

In the example, obj will have two properties myVar and display, where display is a function expression. So, this inside display() method points to obj when calling obj.display().

call() and apply()

In JavaScript, a function can be invoked using () operator as well as call() and apply() method as shown below.

```
function WhoIsThis() {
    alert('Hi');
}

WhoIsThis();
WhoIsThis.call();
WhoIsThis.apply();
```

In the xample, WhoIsThis(), WhoIsThis.call() and WhoIsThis.apply() executes a function in the same way.

The main purpose of call() and apply() is to set the context of this inside a function irrespective whether that function is being called in the global scope or as object's method.

You can pass an object as a first parameter in call() and apply() to which the this inside a calling function should point to.

```
var myVar = 100;

function WhoIsThis() {

    alert(this.myVar);
}

var obj1 = { myVar : 200 , whoIsThis: WhoIsThis };

var obj2 = { myVar : 300 , whoIsThis: WhoIsThis };

WhoIsThis(); // 'this' will point to window object

WhoIsThis.call(obj1); // 'this' will point to obj1

WhoIsThis.apply(obj2); // 'this' will point to obj2

obj1.whoIsThis.call(window); // 'this' will point to
window object

WhoIsThis.apply(obj2); // 'this' will point to obj2
```

As you can see in the above example, when the function WhoIsThis is called using () operator (like WhoIsThis()) then this inside a function follows the rule- refers to window object. However, when the WhoIsThis is called using call() and apply() method then this refers to an object which is passed as a first parameter irrespective of how the function is being called.

Therefore, this will point to obj1 when a function got called as WhoIsThis.call(obj1). In the same way, this will point to obj2 when a function got called like WhoIsThis.apply(obj2)

bind()

The bind() method was introduced since ECMAScript 5. It can be used to set the context of 'this' to a specified object when a function is invoked.

The bind() method is usually helpful in setting up the context of this for a callback function.

```
var myVar = 100;

function SomeFunction(callback)
{
   var myVar = 200;

   callback();
};

var obj = {
         myVar: 300,
         WhoIsThis : function() {
            alert("'this' points to " + this + ", myVar = " +
this.myVar);
         }
    };

SomeFunction(obj.WhoIsThis);
SomeFunction(obj.WhoIsThis.bind(obj));
```

In the example, when you pass obj.WhoIsThis as a parameter to the SomeFunction() then this points to global window object insted of obj, because obj.WhoIsThis() will be executed as a global function by JavaScript engine. You can solve this problem by explicitly setting this value using bind() method. Thus, SomeFunction(obj.WhoIsThis.bind(obj)) will set this to obj by specifying obj.WhoIsThis.bind(obj).

Precedence

So these 4 rules applies to this keyword in order to determine which object this refers to. The following is precedence of order.

- bind()
- call() and apply()
- Object method
- Global scope

So, first check whether a function is being called as callback function using bind()? If not then check whether a function is being called using call() or apply() with parmeter? If not then check whether a function is being called as an object function? Otherise check whether a function is being called in the global scope without dot notation or using window object.

“use strict” Mode

The "use strict" directive was new in ECMAScript version 5.

It is not a statement, but a literal expression, ignored by earlier versions of JavaScript.

The purpose of "use strict" is to indicate that the code should be executed in "strict mode".

With strict mode, you can not, for example, use undeclared variables.

You can use strict mode in all your programs. It helps you to write cleaner code, like preventing you from using undeclared variables.

"use strict" is just a string, so IE 9 will not throw an error even if it does not understand it.

Declaring Strict Mode

Strict mode is declared by adding "use strict"; to the beginning of a script or a function.

Declared at the beginning of a script, it has global scope (all code in the script will execute in strict mode):

```
"use strict";
x = 3.14;       // This will cause an error because x is not declared
```

```
"use strict";
myFunction();

function myFunction() {
  y = 3.14;   // This will also cause an error because y is not
declared
}
```

Declared inside a function, it has local scope (only the code inside the function is in strict mode):

```
x = 3.14;       // This will not cause an error.
myFunction();

function myFunction() {
  "use strict";
  y = 3.14;   // This will cause an error
}
```

The "use strict"; Syntax

The syntax, for declaring strict mode, was designed to be compatible with older versions of JavaScript.

Compiling a numeric literal (4 + 5;) or a string literal ("John Doe";) in a JavaScript program has no side effects. It simply compiles to a non existing variable and dies.

So "use strict"; only matters to new compilers that "understand" the meaning of it.

Why Strict Mode?

Strict mode makes it easier to write "secure" JavaScript.

Strict mode changes previously accepted "bad syntax" into real errors.

As an example, in normal JavaScript, mistyping a variable name creates a new global variable. In strict mode, this will throw an error, making it impossible to accidentally create a global variable.

In normal JavaScript, a developer will not receive any error feedback assigning values to non-writable properties.

In strict mode, any assignment to a non-writable property, a getter-only property, a non-existing property, a non-existing variable, or a non-existing object, will throw an error.

JavaScript Constructor Function

In JavaScript, a constructor function is used to create objects. For example,

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}

// create an object
const person = new Person();
```

In the above example, function Person() is an object constructor function.

To create an object from a constructor function, we use the new keyword.

Create Multiple Objects with Constructor Function

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23,

     this.greet = function () {
        console.log('hello');
    }
}

// create objects
const person1 = new Person();
const person2 = new Person();

// access properties
console.log(person1.name);  // John
console.log(person2.name);  // John
```

JavaScript this Keyword

In JavaScript, when this keyword is used in a constructor function, this refers to the object when the object is created. For example,

```
// constructor function
function Person () {
    this.name = 'John',
}

// create object
const person1 = new Person();

// access properties
console.log(person1.name);  // John
```

Hence, when an object accesses the properties, it can directly access the property as person1.name.

JavaScript Constructor Function Parameters

```
// constructor function
function Person (person_name, person_age, person_gender) {

   // assigning  parameter values to the calling object
    this.name = person_name,
    this.age = person_age,
    this.gender = person_gender,

    this.greet = function () {
        return ('Hi' + ' ' + this.name);
    }
}

// creating objects
const person1 = new Person('John', 23, 'male');
const person2 = new Person('Sam', 25, 'female');

// accessing properties
console.log(person1.name); // "John"
console.log(person2.name); // "Sam"
```

In the example, we have passed arguments to the constructor function during the creation of the object.

```
const person1 = new Person('John', 23, 'male');
const person2 = new Person('Sam', 25, 'male');
```

This allows each object to have different properties. As shown above,

console.log(person1.name); gives John

console.log(person2.name); gives Sam

Adding Properties And Methods in an Object

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}

// creating objects
let person1 = new Person();
let person2 = new Person();

// adding property to person1 object
person1.gender = 'male';

// adding method to person1 object
person1.greet = function () {
    console.log('hello');
}
person1.greet();   // hello
// Error code
// person2 doesn't have greet() method
person2.greet();
```

In the example, a new property gender and a new method greet() is added to the person1 object.

However, this new property and method is only added to person1. You cannot access gender or greet() from person2. Hence the program gives error when we try to access person2.greet();

JavaScript Built-in Constructors

```
let a = new Object();    // A new Object object
let b = new String();    // A new String object
let c = new Number();    // A new Number object
let d = new Boolean();   // A new Boolean object
```

In JavaScript, strings can be created as objects by:

```
const name = new String ('John');
console.log(name); // "John"
```

In JavaScript, numbers can be created as objects by:

```
const number = new Number (57);
console.log(number); // 57
```

In JavaScript, booleans can be created as objects by:

```
const count = new Boolean(true);
console.log(count); // true
```

JavaScript Classes

Classes are one of the features introduced in the ES6 version of JavaScript.

A class is a blueprint for the object. You can create an object from the class.

You can think of the class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions, you build the house. House is the object.

Since many houses can be made from the same description, we can create many objects from a class.

Creating JavaScript Class

JavaScript class is similar to the Javascript constructor function, and it is merely a syntactic sugar.

The constructor function is defined as:

```
// constructor function
function Person () {
    this.name = 'John',
    this.age = 23
}

// create an object
const person1 = new Person();
```

Instead of using the function keyword, you use the class keyword for creating JS classes. For example,

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

The class keyword is used to create a class. The properties are assigned in a constructor function.

Now we can create an object. For example,

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

// creating an object
const person1 = new Person('John');
const person2 = new Person('Jack');

console.log(person1.name); // John
console.log(person2.name); // Jack
```

Here, person1 and person2 are objects of Person class.

Javascript Class Methods

While using constructor function, you define methods as:

```
// constructor function
function Person (name) {

  // assigning  parameter values to the calling
object
   this.name = name;

   // defining method
   this.greet = function () {
      return ('Hello'' + ' ' + this.name);
   }
}
```

It is easy to define methods in JavaScript class. You simply give the name of the method followed by (). For example,

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

    // defining method
    greet() {
        console.log(`Hello ${this.name}`);
    }
}

let person1 = new Person('John');

// accessing property
console.log(person1.name); // John

// accessing method
person1.greet(); // Hello John
```

Getters and Setters

In JavaScript, getter methods get the value of an object and setter methods set the value of an object.

JavaScript classes may include getters and setters. You use the get keyword for getter methods and set for setter methods. For example,

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

    // getter
    get personName() {
        return this.name;
    }

    // setter
    set personName(x) {
        this.name = x;
    }
}

let person1 = new Person('Jack');
console.log(person1.name); // Jack

// changing the value of name property
person1.personName = 'Sarah';
console.log(person1.name); // Sarah
```

Hoisting

A class should be defined before using it. Unlike functions and other JavaScript declarations, the class is not hoisted. For example,

```
// accessing class
const p = new Person(); // ReferenceError

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

class expression

Similar to function expressions, class expressions can be named or unnamed. If named, the name of the class is local to the class body only.

```
const Rectangle = class {
constructor(height, width) {
this.height = height;
 this.width = width;
 }
 area() {
return this.height * this.width;
 }};
console.log(new Rectangle(5, 8).area());// expected output: 40
```

Syntax

```
const MyClass = class [className] [extends otherClassName] {
   // class body
};
```

A simple class expression

```
const Foo = class {
  constructor() {}
  bar() {
    return 'Hello World!';
  }
};

const instance = new Foo();
instance.bar();  // "Hello World!"
Foo.name;        // "Foo"
```

Named class expressions

If you want to refer to the current class inside the class body, you can create a named class expression. The name is only visible within the scope of the class expression itself.

```
const Foo = class NamedFoo {
  constructor() {}
  whoIsThere() {
    return NamedFoo.name;
  }
}
const bar = new Foo();
bar.whoIsThere();  // "NamedFoo"
NamedFoo.name;     // ReferenceError:
NamedFoo is not defined
Foo.name;          // "NamedFoo"
```

Import and Export Declarations

The static import statement is used to import read only live bindings which are exported by another module. Imported modules are in strict mode whether you declare them as such or not. The import statement cannot be used in embedded scripts unless such script has a type="module". Bindings imported are called live bindings because they are updated by the module that exported the binding.

```
import defaultExport from "module-name";

import * as name from "module-name";

import { export1 } from "module-name";

import { export1 as alias1 } from "module-name";

import { export1 , export2 } from "module-name";

import { foo , bar } from "module-name/path/to/specific/un-exported/file";

import { export1 , export2 as alias2 , [...] } from "module-name";

import defaultExport, { export1 [ , [...] ] } from "module-name";

import defaultExport, * as name from "module-name";

import "module-name";
```

defaultExport

Name that will refer to the default export from the module.

module-name

The module to import from. This is often a relative or absolute path name to the .js file containing the module. Certain bundlers may permit or require the use of the extension; check your environment. Only single quoted and double quoted Strings are allowed.

name

Name of the module object that will be used as a kind of namespace when referring to the imports.

exportN

Name of the exports to be imported.

aliasN

Names that will refer to the named imports.

Description

The name parameter is the name of the "module object" which will be used as a kind of namespace to refer to the exports. The export parameters specify individual named exports, while the import * as name syntax imports all of them. Below are examples to clarify the syntax.

Import an entire module's contents

This inserts myModule into the current scope, containing all the exports from the module in the file located in /modules/my-module.js.

```
import * as myModule from '/modules/my-module.js';
```

Here, accessing the exports means using the module name ("myModule" in this case) as a namespace.

Import a single export from a module

Given an object or value named myExport which has been exported from the module my-module either implicitly (because the entire module is exported, for example using export * from 'another.js') or explicitly (using the export statement), this inserts myExport into the current scope.

```
import {myExport} from '/modules/my-module.js';
```

Import multiple exports from module

This inserts both foo and bar into the current scope.

```
import {foo, bar} from '/modules/my-module.js';
```

Examples

We can label any declaration as exported by placing export before it, be it a variable, function or a class.

For instance, here all exports are valid:

```
// export an array
export let months = ['Jan', 'Feb', 'Mar','Apr', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'];

// export a constant
export const MODULES_BECAME_STANDARD_YEAR = 2015;

// export a class
export class User {
  constructor(name) {
    this.name = name;
  }
}
```

```
<script>
// Area function
let area = function (length, breadth) {
        let a = length * breadth;
        console.log('Area of the rectangle is ' + a + ' square
unit');
}

// Perimeter function
let perimeter = function (length, breadth) {
        let p = 2 * (length + breadth);
        console.log('Perimeter of the rectangle is ' + p + '
unit');
}

// Making all functions available in this
// module to exports that we have made
// so that we can import this module and
// use these functions whenever we want.
module.exports = {
        area,
        perimeter
}
</script>
```

Exporting Module Example : library.js

Importing Module Example

For importing any module, use a function called ‘Require’
which takes in the module name and if its user defined module
then its relative path as argument and returns its reference.

The script1.js contains the above JavaScript module (library.js).

```
<script>
// Importing the module library containing
// area and perimeter functions.
// " ./ " is used if both the files are in the same folder.
const lib = require('./library');

let length = 10;
let breadth = 5;

// Calling the functions
// defined in the lib module
lib.area(length, breadth);
lib.perimeter(length, breadth);
</script>
```