

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 WholsThis() {
```

```
    var myVar = 200;
```

```
    alert(myVar); // 200
```

```
    alert(this.myVar); // 100
```

```
}
```

```
WholsThis(); // inferred as window.WholsThis()
```

```
var obj = new WholsThis();
```

```
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 WholsThis() {
    var myVar = 200;

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

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

In the example, a function **WholsThis()** is being called from the global scope. The global scope means in the context of window object. We can optionally call it like **window.WholsThis()**. So in the above example, this keyword in **WholsThis()** 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 **WholsThis()** function.

`var myVar = 100;`       $\longrightarrow$  `== window.myVar`

`function WhoIsThis() {`  
    `var myVar = 200;`  
    `alert(myVar);`  
    `alert(this.myVar);`       $\longrightarrow$  `== window.myVar`  
}

`WhoIsThis();`       $\longrightarrow$  `== window.WhoIsThis()`

'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 WholsThis() {
```

```
        var myVar = 200;
```

```
        alert("myVar = " + myVar); // 200
```

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

```
    }
```

```
    WholsThis();
```

```
}
```

```
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 WholsThis() {
    this.myVar = 200;
}
var obj1 = new WholsThis();

var obj2 = new WholsThis();
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 WholsThis() {
  this.myVar = 200;

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

    alert("myVar = " + myVar); // 300
    alert("this.myVar = " + this.myVar); // 200
  };
}

var obj = new WholsThis();

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.

<pre>function WholsThis() {     alert('Hi'); }  WholsThis(); WholsThis.call(); WholsThis.apply();</pre>	<p>In the example, WholsThis(), WholsThis.call() and WholsThis.apply() executes a function in the same way.</p> <p>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.</p> <p>You can pass an object as a first parameter in call() and apply() to which the this inside a calling function should point to.</p>
---	---

```
var myVar = 100;

function WholsThis() {
    alert(this.myVar);
}

var obj1 = { myVar : 200 , wholsThis: WholsThis };
var obj2 = { myVar : 300 , wholsThis: WholsThis };

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

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

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

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

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

As you can see in the above example, when the function WholsThis is called using () operator (like WholsThis()) then this inside a function follows the rule- refers to window object. However, when the WholsThis 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 WholsThis.call(obj1). In the same way, this will point to obj2 when a function got called like WholsThis.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,
    WholsThis : function() {
        alert("'this' points to " + this + ", myVar = " +
this.myVar);
    }
};

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

In the example, when you pass `obj.WholsThis` as a parameter to the `SomeFunction()` then `this` points to global window object instead of `obj`, because `obj.WholsThis()` 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.WholsThis.bind(obj))` will set `this` to `obj` by specifying `obj.WholsThis.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;  
  }  
}
```