Constructor Function &

this & ***new*** keywords

Outline

- Constructor Function
- ***this*** keyword
- ***new*** keyword

Constructor Function

- In JavaScript, a constructor function is used to create objects.
- To create an object from a constructor function, we use the **new** keyword.
- It is considered a good practice to capitalize the first letter of your constructor function.

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

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

When Constructor Function Invoked

- A new empty object gets created.
- The ***this*** keyword begins to refer to the new object and it becomes the current instance object.
- The new object is then returned as the return value of the constructor.

Constructor Function : Example

```
function Person() {
    this.firstName = "unknown";
    this.lastName = "unknown";
}

var person1 = new Person();
person1.firstName = "Steve";
person1.lastName = "Jobs";

console.log(person1.firstName + " " + person1.lastName);

var person2 = new Person();
person2.firstName = "Bill";
person2.lastName = "Gates";

console.log(person2.firstName + " " + person2.lastName);
```

Add Methods in a Constructor Function

```
function Person() {
    this.firstName = "unknown";
    this.lastName = "unknown";
    this.getFullName = function() {
        return this.firstName + " " + this.lastName;
    }
};

var person1 = new Person();
person1.firstName = "Steve";
person1.lastName = "Jobs";

console.log(person1.getFullName());

var person2 = new Person();
person2.firstName = "Bill";
person2.lastName = "Gates";

console.log(person2.getFullName());
```

Constructor with Parameters

```
function Person(FirstName, LastName, Age) {
    this.firstName = FirstName || "unknown";
    this.lastName = LastName || "unknown";
    this.age = Age || 25;
    this.getFullName = function() {
        return this.firstName + " " + this.lastName;
    }
};

var person1 = new Person("James", "Bond", 50);
console.log(person1.getFullName());

var person2 = new Person("Tom", "Paul");
console.log(person2.getFullName());
```

Constructor vs Object Literal

- An object literal is typically used to create a single object whereas a constructor is useful for creating multiple objects.

```
// using object literal
let person = {
    name: 'Sam'
}
```

```
// using constructor function
function Person () {
    this.name = 'Sam'
}

let person1 = new Person();
let person2 = new Person();
```

Constructor vs Object Literal

- Each object created using a constructor is unique.
- Properties can be added or removed from an object without affecting another one created using the same constructor.

```
// using constructor function
function Person () {
    this.name = 'Sam'
}

let person1 = new Person();
let person2 = new Person();

// adding new property to person1
person1.age = 20;
```

Constructor vs Object Literal

- However, if an object is built using an object literal, any changes made to the variable that is assigned the object value will change the original object.

```
// using object lateral
let person = {
    name: 'Sam'
}

console.log(person.name); // Sam

let student = person;

// changes the property of an object
student.name = 'John';

// changes the origins object property
console.log(person.name); // John
```

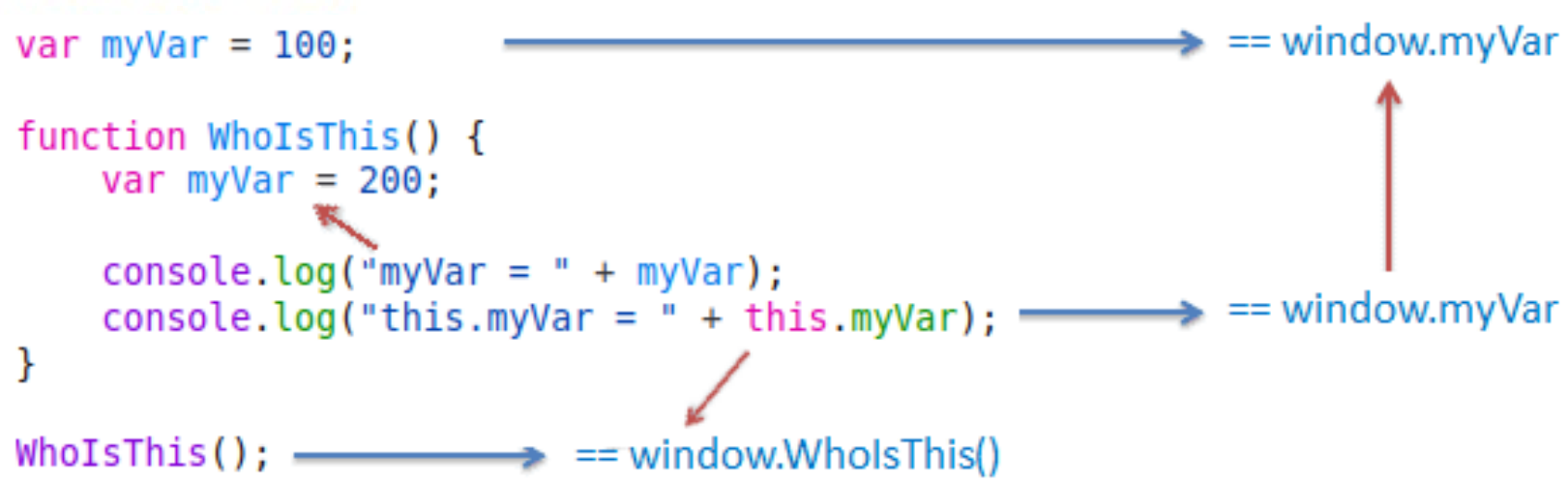
this Keyword

- The ***this*** keyword is one of the most widely used and yet confusing keyword in JavaScript.
- ***this*** points to a particular object. Now, which is that object depends on how a function which includes '***this***' keyword is being called.
- The following four rules apply to this in order to know which object is referred by ***this*** keyword.

1. Global Scope
2. Object's Method
3. call() or apply() method
4. bind() method

this Keyword : Global Scope

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



Output:

```
myVar = 200
this.myVar = 100
```

Note: In the strict mode, value of '**this**' will be undefined inside a function.

this Keyword : Global Scope

- '***this***' points to global window object even if it is used in an inner function.

```
var myVar = 100;

function SomeFunction() {

    function WhoIsThis() {
        var myVar = 200;

        console.log("myVar = " + myVar);
        console.log("this.myVar = " + this.myVar);
    }

    WhoIsThis();
}

SomeFunction();
```

Output:

```
myVar = 200
this.myVar = 100
```

this inside Object's Method – (1)

- 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;

console.log(obj1.myVar);
console.log(obj2.myVar);
```

Output:

```
200

300
```

this inside Object's Method – (2)

```
var myVar = 100;

function WhoIsThis() {
    this.myVar = 200;

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

        console.log("myVar = " + myVar);
        console.log("this.myVar = " + this.myVar);
    };
}
var obj = new WhoIsThis();

obj.display();
```

Output:

```
myVar = 300
this.myVar = 200
```

this inside Object's Method – (3)

```
var myVar = 100;

var obj = {
    myVar: 300,
    whoIsThis: function() {
        var myVar = 200;

        console.log(myVar);
        console.log(this.myVar);
    }
};

obj.whoIsThis();
```

Output:

```
200

300
```

call() and apply()

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

```
function WhoIsThis() {
    console.log('Hi');
}

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

Output:

```
Hi
Hi
Hi
```

call() and apply()

- 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() {

    console.log(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
```

Output:

```
100
200
300
100
300
```

call() vs apply()

- The difference between **call()** and **apply()** is that
  - **call()** passes all arguments after the first one on to the invoked function,
  - **apply()** takes an array as its second argument and passes the members of that array as arguments.
- The following have the same effect.

```
someFunc.call (thisArg, 1, 2, 3)
```

VS

```
someFunc.apply (thisArg, [1, 2, 3])
```

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() {
                console.log("'this' points to " + this + ", myVar = " + this.myVar);
            }
      };

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

Output:

```
'this' points to [object Window], myVar = 100
'this' points to [object Object], myVar = 300
```

Precedence

- These 4 rules apply to this keyword in order to determine which object this refers to. The following is precedence of order.
    1. bind()
    2. call() and apply()
    3. Object’s Method
    4. 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.
- Thus, use these simple rules in order to know which object the 'this' refers to inside any function.

new Keyword

- The new keyword constructs and returns an object (instance) of a constructor function.
- The new keyword performs following four tasks:

1. It creates new empty object e.g. obj = { };
2. It sets new empty object's invisible 'prototype' property to be the constructor function's visible and accessible 'prototype' property.

   Every function has visible 'prototype' property whereas every object includes invisible 'prototype' property (__proto__).

3. It binds property or function which is declared with this keyword to the new object.
4. It returns newly created object unless the constructor function returns a non-primitive value (custom JavaScript object).

   If constructor function does not include return statement then compiler will insert 'return this;' implicitly at the end of the function.

   If the constructor function returns a primitive value then it will be ignored.

new Keyword

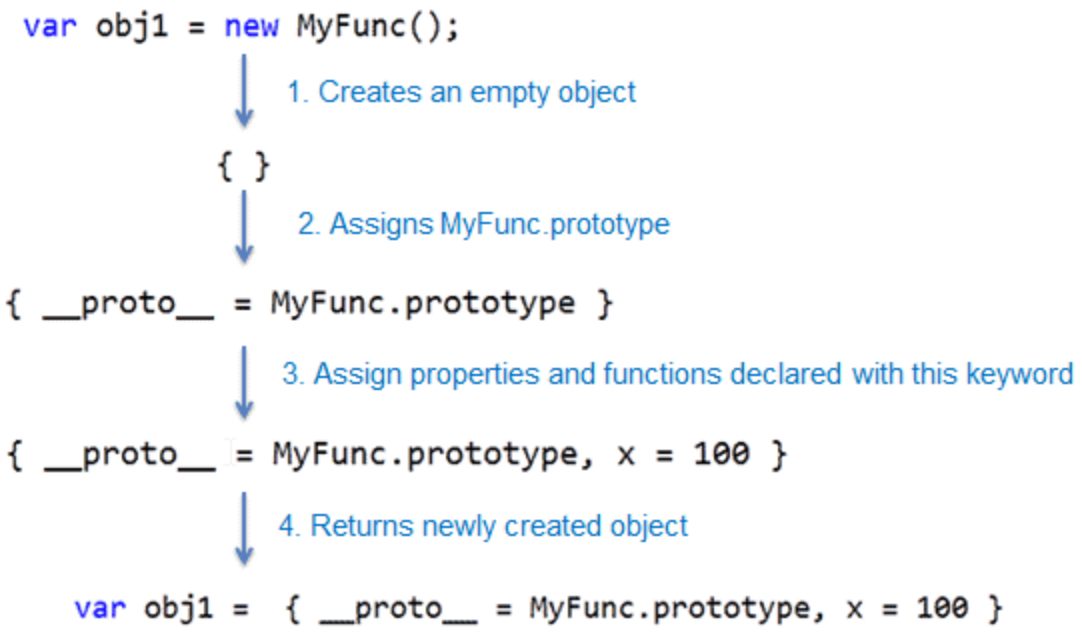
```
function MyFunc() {
    var myVar = 1;
    this.x = 100;
}

MyFunc.prototype.y = 200;

var obj1 = new MyFunc();
console.log(obj1.x);
console.log(obj1.y);
```

Output:

```
100
200
```



new Keyword

- The new keyword ignores return statement that returns primitive value.

```
function MyFunc() {
    this.x = 100;

    return 200;
}

var obj = new MyFunc();
console.log(obj.x);
```

Output:

```
100
```

new Keyword

- If function returns non-primitive value (custom object) then new keyword does not perform above 4 tasks.

```
function MyFunc() {
    this.x = 100;

    return { a: 123 };
}

var obj1 = new MyFunc();

console.log(obj1.x);
```

Output:

```
undefined
```

References

- https://www.tutorialsteacher.com/javascript
- https://www.programiz.com/javascript/constructor-function