Objects

Part 3

Outline

- Accessor Descriptors
- Enumerate the Properties of an Object
- ***this*** keyword
- ***new*** keyword

Accessor Descriptor

- Descriptors for accessor properties are different from those for data properties.
- For accessor properties, there is no value or writable, but instead there are get and set functions.
- That is, an accessor descriptor may have:
  - **get** – a function without arguments, that works when a property is read,
  - **set** – a function with one argument, that is called when the property is set,
  - **enumerable** – same as for data properties,
  - **configurable** – same as for data properties.

Accessor Descriptor

```
let user = {
    name: "John",
    surname: "Smith"
};

Object.defineProperty(user, 'fullName', {
    get() {
        return `${this.name} ${this.surname}`;
    },

    set(value) {
        [this.name, this.surname] = value.split(" ");
    }
});

console.log(user.fullName); // John Smith

for (let key in user)
    console.log(key); // name, surname
```

Accessor Descriptor

- note that a property can be either an accessor (has get/set methods) or a data property (has a value), not both.
- If we try to supply both get and value in the same descriptor, there will be an error:

```
// Error: Invalid property descriptor.
Object.defineProperty({}, 'prop', {
    get() {
      return 1
    },

    value: 2
  });
```

Enumerate the Properties of an Object

There are three native ways to list/traverse object properties:

1. **for...in loop**

   - This method traverses all enumerable properties of an object and its prototype chain

2. **Object.keys(obj)**

   - This method returns an array with all the own (not in the prototype chain) enumerable properties' names ("keys") of an object obj

3. **Object.getOwnPropertyNames(obj)**

   - This method returns an array containing all own properties' names (enumerable or not) of an object obj

for...in Statement

- Iterates over all enumerable properties of an object
  - that are keyed by strings
  - including inherited enumerable properties

```
for (key in object) {
  // executes for each key of object properties
}
```

for...in Statement

```
var user = {
  name: 'John',
  age: 30,
  gender: 'male',
  greet() {
    console.log('hello from ' + this.name)
  }
}
var res = '';
for (let i in user) {
  res = res + i + ' '
}
console.log(res)
//  "name age gender greet "
```

Enumerating Object Properties

```
user = { name: 'John', age: 30 };

Object.defineProperty(user, 'age', { enumerable: false })

Object.keys(user);
// Array [ "name" ]

Object.getOwnPropertyNames(user);
// Array ["name", "age"]
```

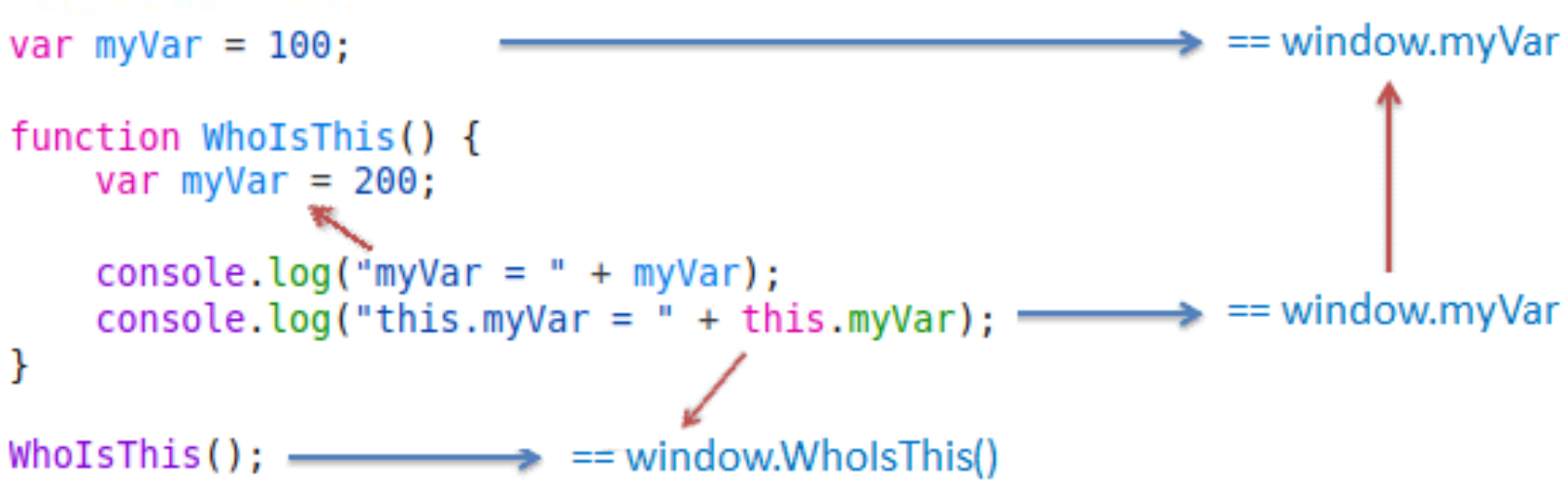
this Keyword

- 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 in the global scope.

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
```

- 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 – (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
```

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
  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

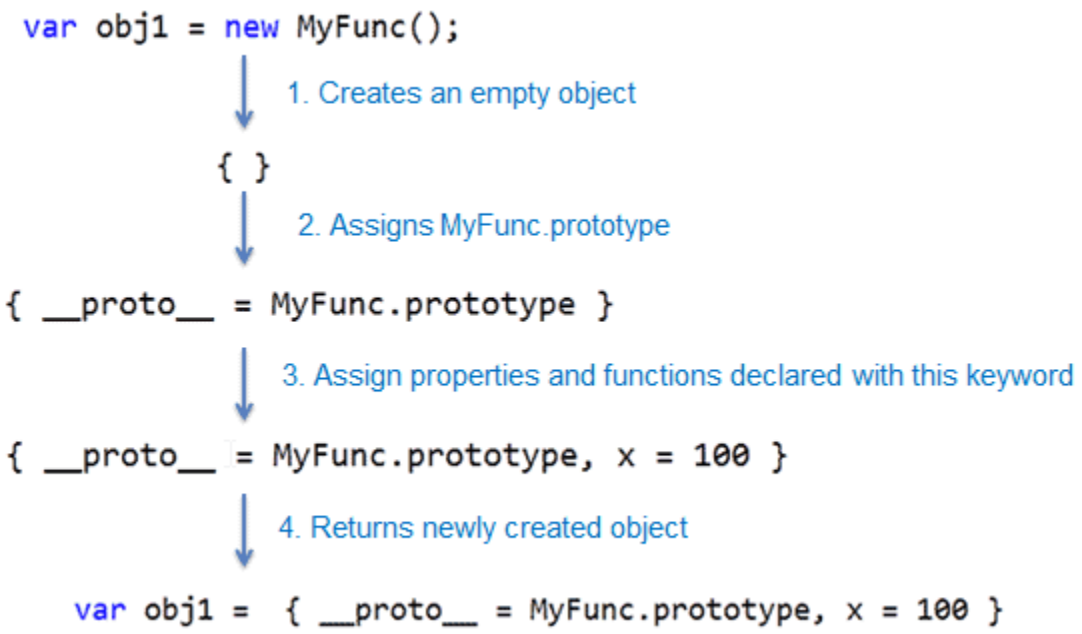
```
function MyFunc() {
    var myVar = 1;
    this.x = 100;
}

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