Advanced JavaScript

ITM - SDM S9

EPITA

Let's go back to last week

```
var Fobject = {
  a: 2,
  foo: function () {
    return "I'm a developer"
  }
}

JSON.stringify(Fobject) // "{"a":2}"
```

Because `function`is not a valid `JSON type`

So we can use `replacer` parameter of `JSON.stringify(...)`

```
JSON.stringify(Fobject, function(key, value) {
  if (typeof value === 'function') {
    return value.toString();
  } else {
    return value;
  }
})
// "{"a":2,"foo":"function () {\n\t\treturn \"I'm a developer\"\n\t}"}"
```

Filter

```
var animals = [
  { name: 'Winnie', species: 'bear'},
  { name: 'Minnie', species: 'mouse'},
  { name: 'Rafiki', species: 'monkey'},
  { name: 'Abu', species: 'monkey'},
  { name: 'Mickey', species: 'mouse'},
  { name: 'Baloo', species: 'bear'},
  { name: 'Piglet', species: 'pig'}
]
```

We want to get all monkeys. What do you do to get that ?

```
var monkeys = [];

for (var i = 0, length = animals.length; i < length; i++) {
  if (animals[i].species == 'monkey') {
    monkeys.push(animals[i]);
  }
}

console.log(monkeys);
/*
[
  {"name":"Rafiki","species":"monkey"},
  {"name":"Abu","species":"monkey"}
]
*/
```

But we can do much simpler with **high order function** `filter`.

`filter` takes a callback function as an arguments.

```
var monkeys = animals.filter(function isMonkey(animals) {
  return animals.species == 'monkey'
});

console.log(monkeys);
/*
[
  {"name":"Rafiki","species":"monkey"},
  {"name":"Abu","species":"monkey"}
]
*/
```

Map

```
var animals = [
  { name: 'Winnie', species: 'bear'},
  { name: 'Minnie', species: 'mouse'},
  { name: 'Rafiki', species: 'monkey'},
  { name: 'Abu', species: 'monkey'},
  { name: 'Mickey', species: 'mouse'},
  { name: 'Baloo', species: 'bear'},
  { name: 'Piglet', species: 'pig'}
]
```

We want to get all animal's names. What do you do to get that ?

```
var names = [];

for (var i = 0, length = animals.length; i < length; i++) {
  names.push(animals[i].name);
}

console.log(names);
/*
["Winnie","Minnie","Rafiki","Abu","Mickey","Baloo","Piglet"]
*/
```

But we can do much simpler with another **high order function** `map`.

`map` takes a callback function as an arguments.

```
var names = animals.map(function getName(animal) {
  return animal.name;
})

console.log(names);

/*
["Winnie","Minnie","Rafiki","Abu","Mickey","Baloo","Piglet"]
*/
```

Property Descriptor

In Javascript all properties are described with property descriptor

```
var myObject = {
    a: 4
};

Object.getOwnPropertyDescriptor( myObject, "a" );
/*
{
    value: 4,
    writable: true,
    enumerable: true,
    configurable: true
}
*/
```

`getOwnPropertyDescriptor` return property descriptor the property descriptor(data-descriptor) `value`, `writable`, `enumerable` and `configurable`

We can add new property or modify an existing one with method `defineProperty`.

Writable

```
var myObject = {};

Object.defineProperty(myObject, "a", {
    value: 2,
    writable: false, // not writable
    configurable: true,
    enumerable: true
});

myObject.a = 3;

myObject.a; // 2
```

It does not raise error

But if you try this in strict mode we get an error

```
'use strict'

var myObject = {};

Object.defineProperty(myObject, "a", {
    value: 2,
    writable: false, // not writable
    configurable: true,
    enumerable: true
});

myObject.a = 3; // TypeError
```

Configurable

As long as property is configurable, we can modify

```
var myObject = {
    a: 2
};

myObject.a = 3;
myObject.a;       // 3

Object.defineProperty( myObject, "a", {
    value: 4,
    writable: true,
    configurable: false,    // not configurable!
    enumerable: true
} );

myObject.a;       // 4
myObject.a = 5;
myObject.a;       // 5

Object.defineProperty( myObject, "a", {
    value: 6,
    writable: true,
    configurable: true,
    enumerable: true
} ); // TypeError
```

The program raise a `TypeError` because we attempt to change the descriptor definition of a non-configurable property.

Be when we change configurable property to `false`, we cannot go back(change to true)

```
var myObject = {
    a: 2
};

myObject.a;                 // 2
delete myObject.a;
myObject.a;                 // undefined

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: false,
    enumerable: true
} );

myObject.a;                 // 2
delete myObject.a;
myObject.a;                 // 2
```

`myObject.a` does not remove because it's non-configurable property so delete failed silenty

Enumarable

We can prevent object to use `for..in` for example, in setting property `enumerable` to `false`

Constant object

We can create object property that cannot be changed, redefined or deleted by setting property `writable: false` and `configurable: false`

```
var myObject = {};

Object.defineProperty( myObject, "FAVORITE_NUMBER", {
    value: 42,
    writable: false,
    configurable: false
} );

myObject["FAVORITE_NUMBER"] = 43;
myObject["FAVORITE_NUMBER"]; // 42
```

Prevent Extension

If we want to create an object that cannot have new properties, we can use `Object.preventExtensions`

```
var myObject = {
    a: 2
};

Object.preventExtensions( myObject );

myObject.b = 3;
myObject.b; // undefined
```

In non-strict mode, the creation of `b` fails silenty.

In strict mode, it throws a `TypeError`.

Seal & Freeze

We can create sealed object, this object cannot add property, cannot reconfigure or delete any existing property but it can still modify their values.

`Object.freeze` creates frozen object, it is the same behavior that `Object.seal()` but we cannot modify properties. This approch is the highest level of immutability.

Getters and Setters

When we want to create or access property object Javascript performed two actions `[[Put]]` and `[[Get]]` respectively.

Let's see how to override `[[Get]]` operation for property object.

There are two ways for realise this:

With object literal syntax:

```
var myObject = {
    // define a getter for `a`
    get a() {
        return 2;
    }
};

myObject.a; // 2
```

And with explicit definition with method `defineProperty`:

```
Object.defineProperty(
    myObject,    // target
    "a",        // property name
    {            // descriptor
        // define a getter for `a`
        get: function(){ return 2 },

        // make sure `a` shows up as an object property
        enumerable: true
    }
);

myObject.a; // 2
```

If we try to set the value of `a` later, the set operation won't throw an error but will just silenty.

Let's see how to override `[[Put]]` operation for property object

```
var myObject = {
    // define a getter for `a`
    get a() {
        return this._a_;
    },

    // define a setter for `a`
    set a(val) {
        this._a_ = val * 2;
    }
};

myObject.a = 2;

myObject.a; // 4
```

When we use getter and setter **the data-descriptor value and writable are ignored**.

Existence

How to check if property exist in object ?

We can use operator `in`:

```
var myObject = {
    a: 2
};

("a" in myObject);                // true
("b" in myObject);               // false
```

Or we can use method `hasOwnProperty`

```
myObject.hasOwnProperty( "a" );    // true
myObject.hasOwnProperty( "b" );    // false
```

Difference between method `hasOwnProperty` and operator `in` is that operator `in` check if property exist in object or if it exist higher level of the [[Prototype]] chain object(we talk about protoype chain later) whereas method `hasOwnProperty` checks to see if only object has a property or not

Enumeration

Go back to enumeration.

A bit earlier, we talk about date-descriptor `enumerable` which means that property object will be included if the object's properties are iterated through

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // make `a` enumerable, as normal
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // make `b` NON-enumerable
    { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty( "b" ); // true

for (var k in myObject) {
    console.log( k, myObject[k] );
}
// "a" 2
```

Enumerable and non-emurable properties can be distinguished by another way:

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // make `a` enumerable, as normal
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // make `b` non-enumerable
    { enumerable: false, value: 3 }
);

myObject.propertyIsEnumerable( "a" ); // true
myObject.propertyIsEnumerable( "b" ); // false

Object.keys( myObject ); // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

Method `propertyIsEnumarable` check if the given property name exist on the object and is also `enumerable:true`

`Object.keys()` returns an array of all enumerable properties whereas `Object.getOwnPropertyNames()` returns an array of all properties, enumerable or not.

THIS

To answer a question: what this `this` a reference to ? We must to determine the call-site: the place in code where a function is called

We must determine which of 4 rules applies I'm going to explain this 4 rules.

First rules: standalone function invocation

```
function foo() {
    console.log( this.a );
}

var a = 2;

foo(); // 2
```

When `foo()` is called, `this.a` resolves to our global variable a because in this case the default binding for this applies to the function call, and so points this at the global object.

Standalone rules is apply when the call-site of function is located with a plain, un-decorated function reference

Implicit Binding

Second rule to consider is: does the call-site have context object

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

The call-site uses the `obj` context to reference the function. The call of `foo()` is preceded by an object reference to `obj`. When there is a context object for function reference. The implicit binding rule says that it's that object which should be used for the function call's this binding.

The last level of an object property reference chain matters to the call-site

```
function foo() {
    console.log( this.a );
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

Implicit lost

Sometimes,the implicity bound function loses this binding which usually means it falls back to the default binding, of either the global object or undefined, depending on strict mode.

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo;  // function reference/alias!

var a = "oops, global"; // `a` also property on global object

bar(); // "oops, global"
```

Even though `bar` appears to be a reference to `obj.foo`, in fact, it's really just another reference to `foo` itself. Moreover, the call-site is what matters, and the call-site is `bar()`, which is a plain, un-decorated call and thus the default binding applies.

Explicit Binding

What if you want to force function call to use a particular object for `this` binding, without putting a property function reference on a object ?

To realise this we can use to function `call(...)` and `apply(...)`. They both take, as their first parameter, an object to use for the `this`, and then invoke the function with that `this` specified

```
function foo() {
    console.log( this.a );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Invoking `foo` with explicit binding by `foo.call(..)` allows us to force its this to be obj.

New binding

In Javascript `constructors` are just a functions that happpen to be called with the `new` in front of them. They are not attached to classes, nor are they instantiating a class. They are not even special types of functions. They're just regular functions that are, in essence, hijacked by the use of new in their invocation.

When a function is invoked with new in front of it, otherwise known as a constructor call, the following things are done automatically:

1. a brand new object is created (aka, constructed) out of thin air
2. the newly constructed object is [[Prototype]]-linked
3. the newly constructed object is set as the this binding for that function call
4. unless the function returns its own alternate object, the new-invoked function call will automatically return the newly constructed object.

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

By calling foo(..) with new in front of it, we've constructed a new object and set that new object as the this for the call of foo(..). So new is the final way that a function call's this can be bound. We'll call this **new binding**.

Mixins

In Javascript, there are no "classes" to instantiate, only object And objects are copied to other object, they are connected together

Other languages class behaviors imply copies.

So JS developpers **fake** the missing copy behavior of Javascript with **mixins**.

There are two types of mixin : **explicit** and **implicit**

Explicit Mixins

We are going to study explicit mixin trough the example: Vehicle and Car

```
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // only copy if not already present
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}

var Vehicle = {
    engines: 1,

    ignition: function() {
        console.log( "Turning on my engine." );
    },

    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log( "Rolling on all " + this.wheels + " wheels!" );
    }
} );
```

We only work object because **there are no class in Javascript** `Car` now has a copy of the properties and function from `Vehicle`

Polymorphism Revisited

Let's see this statement: `Vehicle.drive.call( this )`. This is what call "explicit pseudo-polymorphism"

If we said `Vehicle.drive()`, the `this` binding for that function call would be the `Vehicle` object instead of the `Car`. So, we use `.call(this)`

Because of Javascript characteristic, explicit pseudo-polymorphism creates brittle manual/explicit linkage **in every single function where you need such a (pseudo-) polymorphic reference**.

Pseudo-polymorphic reference is harder to read and harder to maintain code, so ***Explicit pseudo-polymorphism should be avoided wherever possible***

Implicit Mixins

```
var Something = {
    cool: function() {
        this.greeting = "Hello World";
        this.count = this.count ? this.count + 1 : 1;
    }
};

Something.cool();
Something.greeting; // "Hello World"
Something.count; // 1

var Another = {
    cool: function() {
        // implicit mixin of `Something` to `Another`
        Something.cool.call( this );
    }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1 (not shared state with `Something`)
```

`Something.cool.call( this );` call in method `cool` of `Another` "borrow" the function `Something.cool()`and call it in the context of `Another` via `this` binding.

Prototype

All Objects have an internal property denoted [[Prototype]]. It s simply a reference to another object. It is non-null at time of creation.

What's is the [[Prototype]] reference for? Operation `[[Get]]` is invoked when reference a property on an object.

First time, `[[Get]]` checks if object itself has property on it and if yes, it's used

But if not, [[Get]] operation look up in [[Prototype]] link of the object

```
var anotherObject = {
    a: 2
};

// create an object linked to `anotherObject`
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

`Object.create(...)` creates an object with the [[Prototype]] linkage

But if `property` weren't found on `anotherObject`, the [[Prototype]] chain is again consulted and followed

If no matching property is ever found by the end of the chain, [[Get]] return `undefined`

Where does the `[[Prototype]]` chain finish ?

[[Prototype]] chain finishes by the built-in `Object.prototype`.

This object includes common utilities like used by all Javascript objects like `.toString()`, `.valueOf()` and more.