**Object-Oriented Design Patterns**

**Introduction**

In this lesson, we’ll take a closer look at a few design patterns to create a new object and extend an object (without involving its prototype).

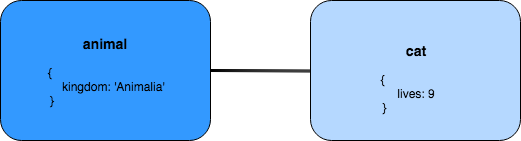
We’ll look at *mixins -* what they are, and how they can be used to copy properties from one object to another object.

Then, we’ll create objects with private properties using two different design patterns - the Module Pattern and Revealing Module Pattern.

**Mixins / Extending Object Functionality with Mixins**

**An Object is Prototype-linked to a Single Object**

Recall that an object’s.prototype property points to just one object. This is because JavaScript only supports single inheritance. If there is an object A and an object B, object C can only be prototype-linked to either A or B.



**Mixins**

In JavaScript an object can only be prototype-linked to a single object.

So, how can we go about extending properties and methods from multiple different sources? A mixin allows us to do that!

A *mixin* is a technique that takes the properties and methods from one object and copies them over to another object. In other words: a technique that provides some useful functionality, but it is not meant to be added to the prototype chain.

**Object.assign()**

The simplest way to implement a mixin pattern is to use *Object.assign()*.

Object.assign() is a method that adds to the target object by merging in the source object(s):

/\* Mixin \*/

let target = { letter: 'a', number: 11 };

let source = { number: 7 };

Object.assign(target, source);

console.log(target);

// { letter: 'a', number: 7 }

The first argument passed in, target, is the destination that receives the properties copied from the source object, source.

Note that Object.assign() does not create and return a new object; it directly modifies then returns the same target object that was passed in! As such, values of existing properties will be overwritten, while properties that don't exist in the source object will remain intact.

**Multiple Source Objects**

*Object.assign()* can even take in multiple different source objects:

const duck = {

hasBill: true

};

const beaver = {

hasTail: true

};

const otter = {

hasFur: true,

feet: 'webbed'

};

const platypus = Object.assign({}, duck, beaver, otter);

console.log(platypus);

// { hasBill: true, hasTail: true, hasFur: true, feet: 'webbed' }

After merging an empty target object (i.e., an object without properties of its own) with the properties from duck, beaver, and otter, the target object is returned with all four properties. It is important to note that the platypus object is not prototype-linked to the three other objects! That is, platypus doesn't exist in any of the three source objects' prototype chains, and vice versa:

console.log(platypus.constructor);

// Object()

console.log(platypus.isPrototypeOf(duck));

// false

console.log(duck.isPrototypeOf(platypus));

// false

console.log(platypus.isPrototypeOf(beaver));

// false

console.log(beaver.isPrototypeOf(duck));

// false

console.log(platypus.isPrototypeOf(otter));

// false

console.log(otter.isPrototypeOf(platypus));

// false

**Functional Mixins**

**Remember Constructor Functions?**

We previously used a constructor function to create a new object, and the *new* keyword to instantiate it:

function City(name, population) {

this.name = name;

this.population = population;

this.identify = function () {

console.log(`${this.name}'s population is ${this.population}.`);

};

}

const sanFrancisco = new City('San Francisco', 870000);

console.log(sanFrancisco);

const mountainView = new City('Mountain View', 78000);

console.log(mountainView);

The same constructor can be instantiated multiple times to create objects.

**Factory Functions**

A factory function is a function that returns an object, but isn’t itself a class or constructor. As such, we invoke a factory function as a normal function without using the *new* operator.

Using a factory function, we can easily create objects instances without the complexity of classes and constructors!

function Basketball(color) {

return {

color: color,

numDots: 35000

};

// this.color = color

// this.numDots = 35000

}

// const orangeBasketball = new Basketball('orange');

const orangeBasketball = Basketball('orange');

console.log(orangeBasketball);

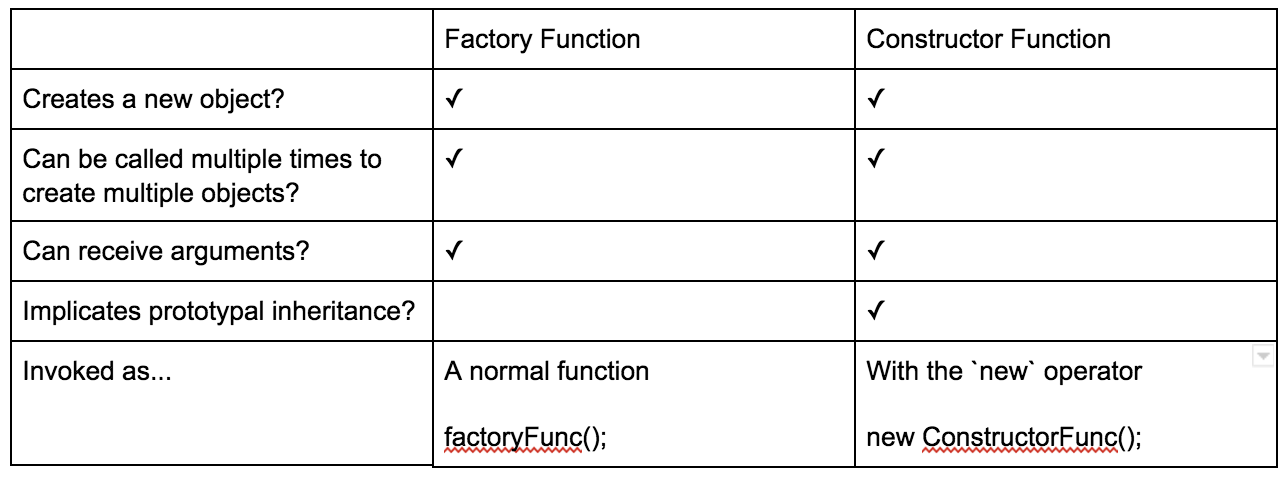
// { color: 'orange', numDots: 35000 }

const myBB = Basketball('blue and green');

const yourBB = Basketball('purple');

const bouncy = Basketball('neon pink');

Comparing and contrasting factory functions and constructor functions:



Factory functions can return a closure - combination of a function and lexical environment (local variables) within which that function was declared. Like an instance of a class, the closure has its own functions and variables as well.

function Radio(mode) {

let on = false;

return {

mode: mode,

turnOn: function () {

on = true;

},

isOn: function () {

return on;

}

};

}

let fmRadio = Radio('fm');

fmRadio.on;

//undefined

console.log(fmRadio.isOn());

// false

console.log(fmRadio.turnOn());

console.log(fmRadio.isOn());

//true

By using closure, we can preserve the state of the lexical environment.

**Functional Mixins**

A functional mixin is a composable factory function that receives a \_mixins\_ as an argument, copies properties and methods from that mixin, and returns a new object.

function CoffeeMaker(object) {

let needsRefill = false;

return Object.assign({}, object, {

pourAll: function () {

needsRefill = true;

},

isEmpty: function () {

return needsRefill;

}

});

}

Unlike a standard factory function, that takes in individual property values as arguments, the functional mixin actually takes in an object itself.

Whichever object is passed into the function, is merged with other objects passed into *Object.assign()*.

function CoffeeMaker(object) {

let needsRefill = false;

return Object.assign({}, object, {

pourAll: function () {

needsRefill = true;

},

isEmpty: function () {

return needsRefill;

}

});

}

const mixedCoffeeMaker = CoffeeMaker({style: 'percolator'});

console.log(mixedCoffeeMaker);

/\*

{

style: 'percolator',

pourAll: function () {

needsRefill = true;

},

isEmpty: function () {

return needsRefill;

}

}

\*/

**The Module Pattern**

**Private Properties: Literal**

By default, most things are publicly accessible in JavaScript.

We can use closure to make certain parts of an app private, but what if we want to prevent access to a property directly?

**No Private Property**

Since JavaScript has no concept of private property out of the box, there is no special syntax or keyword we can use to protect certain properties from being accessed.

However, there is a hope:

**Closure**provides a way to create private data.

/\* closure and scope \*/

function instantiateDeveloper2() {

let name = 'Veronika'

return {

getName: function () {

return this.name;

}

};

}

let developer2 = instantiateDeveloper2();

console.log(developer2.name, developer.getName());

//undefined Veronika

**The Module Pattern**

The Module Pattern leverages many of the powerful features of JavaScript, such as scope, closures, and immediately-invoked function expressions (IIFE).

To create a Module Pattern, we need to:

1. Create a IIFE with a return object.
2. Create a local variable inside this IIFE (using let keyword to declare) - but outside the returned object.
3. Add methods to the returned object.
4. Methods inside the returned object retain their parent scope, which means they can refer to the local variable name inside their parent.

Example:

/\* Module Pattern \*/

let person = (function () {

let name = 'Veronika';

return {

getName: function () {

return name;

},

setName: function (myName){

name = myName;

}

};

})();

console.log(person.name);

// undefined

console.log(person.getName());

// 'Veronika'

person.setName('Not Veronika');

//

console.log(person.name);

// undefined

console.log(person.getName());

// 'Not Veronika'

The Module Pattern uses closures to create private and public properties of an object.

**Other Benefits of the Module Pattern**

More than creating private properties in JavaScript, the Module Pattern can also help with organisation, by partitioning code and providing structure as an application scales.

**The Revealing Module Pattern**

Revealing Module Pattern is a slight variation to the Module Pattern.

With the Revealing Module Pattern, all data and functions are private and only the ones that developers want to be public are revealed.

The underlying philosophy of the Revealing Module Pattern is that, while we still maintain encapsulation (as in the Module Pattern), we also reveal certain properties (and methods).

1. An IIFE (wrapper)
2. The module content (variables, methods, objects, etc.)
3. A returned object literal

/\* Revealing Module Pattern \*/

let myModule = (function (){

function privateMethod (message) {

console.log(message);

}

function publicMethod (message) {

privateMethod(message);

}

return {

publicMethod: publicMethod

};

})();

console.log(Object.keys(myModule))

console.log(myModule.publicMethod("Hello"))

**Another Example**

/\* Another Example \*/

let person = (function () {

let privateAge = 0;

let privateName = 'Andrew';

function privateAgeOneYear() {

privateAge += 1;

console.log(`One year has passed! Current age is ${privateAge}`);

}

function displayName() {

console.log(`Name: ${privateName}`);

}

function ageOneYear() {

privateAgeOneYear();

}

return {

name: displayName,

age: ageOneYear

};

})();

**Benefits of Revealing Module Pattern**

For one, there is a clarity at the end of the module (i.e., the return statement) as to which variables or methods may be accessed publicly.

Revealing Module Pattern also uses a consistent syntax.