Destructuring in JavaScript

Destructuring assignment allows you to assign the properties of an array or object to variables using syntax that looks similar to array or object literals.

TypeScript supports the following forms of Destructuring (literally named after de-structuring i.e. breaking up the structure.

1. **Object Destructuring**
2. **Array Destructuring**

It is easy to think of destructuring as an inverse of structuring. The method of structuring in JavaScript is the object literal

**Object Destructuring**

Destructuring is useful because it allows you to do in a single line, what would otherwise require multiple lines.

Example

var rect = { x: 0, y: 10, width: 15, height: 20 };

var {x, y, width, height} = rect;

console.log(x, y, width, height); // 0,10,15,20

rect.x = 10;

({x, y, width, height} = rect); // assign to existing variables using outer parentheses

console.log(x, y, width, height); // 10,10,15,20

Here in the absence of destructuring you would have to pick off x,y,width,height one by one from rect.

**Object Destructuring with rest**

You can pick up any number of elements from the an object and get an object of the remaining elements using object destructuring with rest.

var {w, x, ...remaining} = {w: 1, x: 2, y: 3, z: 4};

console.log(w, x, remaining); // 1, 2, {y:3,z:4}

**Array Destructuring**

function getDetail(){

return ['Samsung',2017,14000];

};

var [name,release,price] = getDetail();

console.log(name);

console.log(release);

console.log(price);

console.log(getDetail());

**Array Destructuring with rest**

You can pick up any number of elements from the array and get an array of the remaining elements using array destructuring with rest.

var [x, y, ...remaining] = [1, 2, 3, 4];

console.log(x, y, remaining); // 1, 2, [3,4]

**Array Destructuring with ignores**

You can ignore any index by simply leaving its location empty i.e. , , in the left hand side of the assignment. For example:

var [x, , ...remaining] = [1, 2, 3, 4];

console.log(x, remaining); // 1, [3,4]

Map in JavaScript

The Map object holds key-value pairs. Any value (both objects and [primitive values](https://developer.mozilla.org/en-US/docs/Glossary/Primitive)) may be used as either a key or a value.

A Map object iterates its elements in insertion order — a [for...of](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for...of) loop returns an array of [key, value] for each iteration.

Here an example that shows how we can create a **Map** object and how to iterate a map.

var myMap = new Map();

myMap.set(55,'Double five');

myMap.set(45,'Four five');

myMap.set(00,'Double zero');

for(let [key,value] of myMap)

{

console.log(‘keys are '+key+' Values is '+value);

}

**Methods in Map**

[**Map. clear()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/clear)

Removes all key/value pairs from the Map object.

[**Map. delete(key)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/delete)

Returns true if an element in the Map object existed and has been removed, or falseif the element does not exist. Map. has(key) will return false afterwards.

[**Map. entries()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/entries)

Returns a new Iterator object that contains an array of [key, value] for each element in the Map object in insertion order.

[**Map.get(key)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/get)

Returns the value associated to the key, or undefined if there is none.

[**Map.has(key)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/has)

Returns a boolean asserting whether a value has been associated to the key in the Mapobject or not.

[**Map. keys()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/keys)

Returns a new Iterator object that contains the keys for each element in the Map object in insertion order.

[**Map.set(key, value)**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/set)

Sets the value for the key in the Map object. Returns the Map object.

[**Map. values()**](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map/values)

Returns a new Iterator object that contains the values for each element in the Map object in insertion order.

Set in JavaScript

The Set object lets you store unique values of any type, whether [primitive values](https://developer.mozilla.org/en-US/docs/Glossary/Primitive) or object references.

Set objects are collections of values. You can iterate through the elements of a set in insertion order. A value in the Set **may only occur once**; it is unique in the Set's collection.

Here is an example for set

var birds = new Set();

birds.add('NightAngle');

birds.add('Eagle');

birds.add('Parrot');

birds.add('Crow');

birds.add('Eagle');

for(let key of birds)

{

console.log('keys is '+key);

}console.log(birds.size);

**Methods in Set**

[**has(value)**](http://www.collectionsjs.com/method/has-value)

Whether an equivalent value exists in this set .

[**get(value)**](http://www.collectionsjs.com/method/get-value)

Retrieves the equivalent value from the set.

[**add(value)**](http://www.collectionsjs.com/method/add-value)

Adds a value to a set.

[**delete(value)**](http://www.collectionsjs.com/method/delete-value)

Deletes the first equivalent value. Returns whether the key was found and successfully deleted.

[**clear()**](http://www.collectionsjs.com/method/clear)

Deletes all of the values in the set.

Design Pattern

# Module Design Pattern

JavaScript modules are the most prevalently used design patterns for keeping particular pieces of code independent of other components. This provides loose coupling to support well-structured code. Modules should be Immediately-Invoked-Function-Expressions (IIFE) to allow for private scopes - that is, a closure that protect variables and methods (however, it will return an object instead of a function).

Syntax:-

(function() {

// declare private variables and/or functions

return {

// declare public variables and/or functions

}

})();

Example

var HTMLChanger = (function() {

var contents = 'contents'

var changeHTML = function() {

var element = document.getElementById('attribute-to-change');

element.innerHTML = contents;

}

return {

callChangeHTML: function() {

changeHTML();

console.log(contents);

}

};

})();

HTMLChanger.callChangeHTML(); // Outputs: 'contents'

console.log(HTMLChanger.contents);

# Revealing Module Pattern

A variation of the module pattern is called the **Revealing Module Pattern**. The purpose is to maintain encapsulation and reveal certain variables and methods returned in an object literal. The direct implementation looks like this:

# [Singleton](https://scotch.io/bar-talk/4-javascript-design-patterns-you-should-know#toc-singleton)

A Singleton only allows for a single instantiation, but many instances of the same object. The Singleton restricts clients from creating multiple objects, after the first object created, it will return instances of itself

var printer = (function () {

var printerInstance;

function create () {

function print() {

// underlying printer mechanics

}

function turnOn() {

// warm up

// check for paper

}

return {

// public + private states and behaviors

print: print,

turnOn: turnOn

};

}

return {

getInstance: function() {

if(!printerInstance) {

printerInstance = create();

}

return printerInstance;

}

};

function Singleton () {

if(!printerInstance) {

printerInstance = intialize();

}

};

})();

The create method is private because we do not want the client to access this, however, notice that the *getInstance* method is public. Each officer worker can generate a printer instance by interacting with the *getInstance* method, like so:

var officePrinter = printer.getInstance();