**BUILT-IN**

**SYMBOL:**

Q: What’s that for?

A: To avoid confusion

Ex:

**const** bowl = {

'apple': { color: 'red', weight: 136.078 },

'banana': { color: 'yellow', weight: 183.15 },

'orange': { color: 'orange', weight: 170.097 }

};

The bowl contains fruit which are objects that are properties of the bowl. But, we run into a problem when the second banana gets added.

**const** bowl = {

'apple': { color: 'red', weight: 136.078 },

'banana': { color: 'yellow', weight: 183.151 },

'orange': { color: 'orange', weight: 170.097 },

'banana': { color: 'yellow', weight: 176.845 }

};

console.log(bowl);

Object {apple: Object, banana: Object, orange: Object}

Instead of adding another banana to the bowl, our previous banana is overwritten by the new banana being added to the bowl. To fix this problem, we can use symbols.

**const** bowl = {

[Symbol('apple')]: { color: 'red', weight: 136.078 },

[Symbol('banana')]: { color: 'yellow', weight: 183.15 },

[Symbol('orange')]: { color: 'orange', weight: 170.097 },

[Symbol('banana')]: { color: 'yellow', weight: 176.845 }

};

console.log(bowl);

Object {Symbol(apple): Object, Symbol(banana): Object, Symbol(orange): Object, Symbol(banana): Object}

By changing the bowl’s properties to use symbols, each property is a unique Symbol and the first banana doesn’t get overwritten by the second banana.

**ITERATION & ITERATABLE PROTOCOL:**

\_ Iterator method is a zero arguments function that returns an iterator object. Used via [Symbol.iterator]

\_ Iterator object is an object that conforms to the iterator protocol

\_ Iterator protocol means how an object products a sequence of value

ex:

const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

const arrayIterator = digits[Symbol.iterator]();

console.log(arrayIterator.next());

console.log(arrayIterator.next());

console.log(arrayIterator.next());

Object {value: 0, done: false}

Object {value: 1, done: false}

Object {value: 2, done: false}

… (and so on)

// .next() is a method that returns the value of that next value in the sequence. Done means if the iterator has reached the end of its sequence of value

Q: Git push vs Git push origin vs Git push origin master?

A: Git push -> push the current branch to corresponding remote branches

Git push origin === Git push (other than master)

Git push origin master -> only push to master

🡪 Check this answer again

**SETS:**

\_An array or object that contains distinct, non-duplicate items

Ex: nums = [2,4,5,7]

Q: How to create a SET?

A:

1. Empty set:

const games = new Set();

console.log(games);

// print: Set{}

1. Create a Set with a list of vaues

Const games = new Set ([‘Super Mario’ , ‘Pokemon’, ‘Captain Tsubasa’, ‘Pokemon’]

//Set {‘ Super Mario’, ‘Captain Tsubasa’, ‘Pokemon’}

* It deletes the duplicate

A set: not index-based: you do not refer to items in a set based on their position in the set

+) Also items in a Set can’t be accessed individually

Q: How to add or delete an item in a SET?

A: .add() and .delete();

const games = new Set(['Super Mario Bros.', 'Banjo-Kazooie', 'Mario Kart', 'Super Mario Bros.']);

games.add('Banjo-Tooie');

games.add('Age of Empires');

games.delete('Super Mario Bros.');

console.log(games);

//Set {'Banjo-Kazooie', 'Mario Kart', 'Banjo-Tooie', 'Age of Empires'}

Q: if I want to delete every item in the Set?

A: .clear()

Ex: games.clear();

// print: Set {}

Q: Check length of Sets?

A: .size()

Q: Check if an item exists in Set?

A: set.has(item)

//true if yes

//false if no

Q: To retrieve all the values?

A:

Finally, use the .values() method to return the values in a Set. The return value of the .values() method is a SetIterator object.

console.log(months.values());

//SetIterator {'January', 'February', 'March', 'April', 'May', 'June', 'July', 'August', 'September', 'October', 'November', 'December'}

Q: To loop through a SET?

A: 2 ways:

1. Use iterator.next() until it returns true
2. Use the for… of loop:

const colors = new Set(['red', 'orange', 'yellow', 'green', 'blue', 'violet', 'brown', 'black']);

for (const color of colors) {

console.log(color);

}

**MAPS:**

Q: What is Map?

A: contains key-value pairs as opposed to Set only containing values

Map is like Objects

Set is like Arrays

Ex:

const employees = new Map();

employees.set('james.parkes@udacity.com', {

firstName: 'James',

lastName: 'Parkes',

role: 'Content Developer'

});

employees.set('julia@udacity.com', {

firstName: 'Julia',

lastName: 'Van Cleve',

role: 'Content Developer'

});

employees.set('richard@udacity.com', {

firstName: 'Richard',

lastName: 'Kalehoff',

role: 'Content Developer'

});

console.log(employees);

**PROMISES:**

Q: What is it?

A: It will let you do some work that would be done asynchronously and let you get back to your regular work

+) When you create the promise, you must give it the code that will be run asynchronously

+) You give that code as the argument of the constructor function (new Promise() )

Ex:

new Promise(function () {

window.setTimeout(function createSundae(flavor = 'chocolate') {

const sundae = {};

// request ice cream

// get cone

// warm up ice cream scoop

// scoop generous portion into cone!

}, Math.random() \* 2000);

});

// This code creates a promise that will start in a few seconds after I make the request. Then there are a number of steps that need to be made in the createSundae function.

// setTimeOut is the code run asynchronously

Q: How to indicate a successful or a failed request?

A: by using resolve and reject

Ex:

new promise (function (resolve, reject) {

window.setTimeout(function createSundae(flavor = ‘coconut’) {

const sundae ={};

// request ice cream

// get cone

//warm up the ice cream scoop

// scoop ice cream into cone

if( /\*iceCreamConeIsEmpty(flavor) \*/) {

reject(‘ Sorry we’re out of that flavor) ;

}

resolve(sundae);

}, Math.random() \* 2000;

});

// if there’s an error, return a value, in this case a message

// otherwise the resolve method will return the data we want to pass in. It is used to indicate that the request is completed successfully

**PROMISES RETURN IMMEDIATELY**

\_Promise will immediately return an object

const myPromiseObj = new Promise(function (resolve, reject) {

// sundae creation code

});

there’s also a .then() method to notify us if the request we made succeeded or failed

.then() takes 2 functions:

1. The function to run if the request completed successfully
2. The function to run if the request failed to complete

Ex:

const myPromiseObj = new Promise(function (resolve, reject) {

mySundae.then(function(sundae) {

console.log(`Time to eat my delicious ${sundae}`);

}, function(msg) {

console.log(msg);

self.goCry(); // not a real method

})

});

//The first function in green is for successful (resolve)

// The second function is reject

// Just a different way to write Promises. Can use it without then()

Q: When to use Promises with then()?

A:

**PROXIES:**

\_An object that stands in for another object

\_Created via new Proxy(): . This constructor takes 2 items:

+) The object it will be the proxy for

+) An object that contain the list of methods for the proxied objects

Ex:

var richard = {status: 'looking for work'};

var agent = new Proxy(richard, {});

agent.status; // returns 'looking for work'