ES6 Built-ins Udacity Full

1. Symbols
2. The Iterable Protocol
3. A Set
4. WeakSet
5. Maps
6. WeakMaps
7. Promises
8. Proxies
9. Generators
0. ES6 Syntax Deploy

Symbols

A symbol is a `unique` and **immutable data type** that is often used to identify object properties.

To create a symbol, you write Symbol() with an optional string as its description.

```
const sym1 = Symbol('apple');

console.log(sym1);

Symbol(apple)
```

This will create a unique symbol and store it in sym1. The description "apple" is just a way to describe the symbol, but it can't be used to access the symbol itself.

And just to show you how this works, if you compare two symbols with the same description...

```
const sym2 = Symbol('banana');

const sym3 = Symbol('banana');

console.log(sym2 === sym3);

false
```

...then the result is false because the description is only used to described the symbol. It's not used as part of the symbol itself—each time a new symbol is created, regardless of the description.
Adding new property of the object, while there is a proeprty with the same name: (the first one will be overwritten by the second one.)

```
const bowl = {
```

```
  'apple': { color: 'red', weight: 136.078 },
  'banana': { color: 'yellow', weight: 183.15 },
  'orange': { color: 'orange', weight: 170.097 }
}
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.

The Iterable Protocol

The iterable protocol is used for defining and **customizing the iteration behavior of objects.**

How it Works
In order for an object to be iterable, it must implement the **iterable interface**. If you come from a language like Java or C, then you're probably familiar with interfaces, but for those of you who aren't, that basically

means that in order for an **object to be iterable it must contain a default iterator method**. This method will define how the object should be iterated.

The iterator method, which is available via the constant [`Symbol.iterator`], is a **zero arguments function that returns an iterator object**. An iterator object is an object that conforms to the iterator protocol.

The Iterator Protocol

The iterator protocol is used to define a standard way that an object produces a sequence of values. What that really means is **you now have a process for defining how an object will iterate**. This is done through implementing the .next() method.

How it Works
An object becomes an iterator when it implements the `.next()` *method.* The .next() method is a zero arguments function that returns an object with two properties:

- value : the data representing the next value in the sequence of values within the object
- done : a boolean representing if the iterator is done going through the sequence of values
- If done is true, then the iterator has reached the end of its sequence of values.
- If done is false, then the iterator is able to produce another value in its sequence of values.

```
//So here when we are iterating throughthe array, from the next() method, we get an object with
 the current value of the iterated object and if is the last element or not. last=true
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
const arrayIterator = digits[Symbol.iterator]();

console.log(arrayIterator.next());        //Object {value: 0, done: false}
console.log(arrayIterator.next());        //Object {value: 1, done: false}
console.log(arrayIterator.next());        //Object {value: 2, done: false}
```

/ Programming Quiz: Make An Iterable Object**
*****
*** Turn the `james` object into an iterable object.**
*****
*** Each call to iterator.next should log out an object with the following info:**
*** - key: the key from the `james` object**
*** - value: the value of the key from the `james` object**
*** - done: true or false if there are more keys/values**
*****
*** For clarification, look at the example console.logs at the bottom of the code.**
*****
*** Hints:**
*** - Use `Object.keys()` to store the object's properties in an array.**
*** - Each call to `iterator.next()` should use this array to know which property to return.**
*** - You can access the original object using `this`.**
*** - To access the values of the original object, use `this` and the key from the `Object.keys()` array.**

*/

```
// This is a way to iterate over an object.
//Sowe have an interation protocoll, Symbol.iterator which has to be a no argument function, wh
ich returns an ojbect,
//Since we are iterating the object, we can select the current iteration with the this keyword.
// This has an inner function of next:(). which is actualll returning an object of the desired
 values
const james = {
    name: 'James',
    height: `5'10"`,
    weight: 185
};
james[Symbol.iterator] = function () {
    let keys = Object.keys(this);                    //Refers the key of current iteration thro
ugh the object
    let index = 0;
    return {
        next: () => {
            return {
                key: keys[index], value: this[keys[index]], done: ++index >= keys.length
            };
        }
    };
};

const iterator = james[Symbol.iterator]();
console.log(iterator.next().value); // 'James'
console.log(iterator.next().value); // `5'10`
console.log(iterator.next().value); // 185
```

A Set

If you think back to mathematics, a set is **a collection of distinct items**. For example, {2, 4, 5, 6} is a set because each number is unique and appears only once.
Basically, a Set is an object that lets you store unique items. You can add items to a Set, remove items from a Set, and loop over a Set. These items can be either primitive values or objects.

- Sets are not indexed-based - you do not refer to items in a set based on their position in the set
- items in a Set can't be accessed individually

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

console.log(games);

//Set {'Super Mario Bros.', 'Banjo-Kazooie', 'Mario Kart'}
```

Notice the example above automatically removes the duplicate entry "Super Mario Bros." when the Set is created.

Functions of set

- games.add('Banjo-Tooie'); -->Returns the `Set` if an item is successfully added, doesn'tcreate duplicates or provide false message
- games.delete('Super Mario Bros.'); -->Returns a Boolean (true or false) depending on successful deletion.
- games.clear() --->Delete all of theitems from the set
- games.size() --->It is not like indexarrays, so you can't use .lenght property
- games.has() --->Check the existence in the set, true if yes, false if not
- games..values() --->Returns a `SetIterator` object. SetIterator {'Super Mario Bros.', 'Banjo-Kazooie', 'Mario Kart'}

Looping sets:

1. Using the SetIterator (buiut in iteration method)

```
const iterator = games.values();

iterator.next();          //Object {value: 'Super Mario Bros.', done: false}

iterator.next();          //Object {value: 'Banjo-Kazooie', done: false}
```

2. Using a for...of Loop
An easier method to loop through the items in a Set is the for...of loop.

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

for (const color of colors) {
```

```
  console.log(color);
}
```

WeakSet

A WeakSet is just like a normal Set with a few key differences:

- a WeakSet can only contain objects
- a WeakSet is not iterable which means it can't be looped over
- a WeakSet does not have a .clear() method
- You can create a WeakSet just like you would a normal Set, except that you use the WeakSet constructor.

```
const student1 = { name: 'James', age: 26, gender: 'male' };

const student2 = { name: 'Julia', age: 27, gender: 'female' };

const student3 = { name: 'Richard', age: 31, gender: 'male' };


const roster = new WeakSet([student1, student2, student3]);

console.log(roster);

//WeakSet {Object {name: 'Julia', age: 27, gender: 'female'}, Object {name: 'Richard', age: 31,

 gender: 'male'}, Object {name: 'James', age: 26, gender: 'male'}}
```

...but if you try to add something other than an object, you'll get an error!

```
roster.add('Amanda');

Uncaught TypeError: Invalid value used in weak set(…)
```

Garbage Collection
In JavaScript, memory is allocated when new values are created and is "automatically" freed up when those values are no longer needed. This process of freeing up memory after it is no longer needed is what is known as garbage collection.

WeakSets take advantage of this by exclusively working with objects. If you set an object to null, then you're essentially deleting the object. And when JavaScript's garbage collector runs, the memory that object previously occupied will be freed up to be used later in your program.

```
//So basically, it is a collection of object references, and if the garbage collection procedur
e can't find the object, it will remove it from the weakset.
student3 = null;
console.log(roster);
WeakSet {Object {name: 'Julia', age: 27, gender: 'female'}, Object {name: 'James', age: 26, gen
der: 'male'}}
```

Maps

If Sets are similar to Arrays, then Maps are similar to Objects because **Maps store key-value pairs** similar to how objects contain named properties with values.
Essentially, a Map is an object that lets you store key-value pairs **where both the keys and the values can be objects**, primitive values, or a combination of the two.

Methods:

- map.set(key,value) --->Object with the same key will overwrite the existing key-value pair
- map.delete(key) --->Returns true if successful, false if not
- map.clear() --->Delete all of key-value pairs
- map.has() --->Check existence true=yes, false=no
- map.get() --->Retrieve objects with the key

```
//Create map
const employees = new Map();
//Add key-value pairs
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);        //Map {'james.parkes@udacity.com' => Object {...}, 'julia@udacit
y.com' => Object {...}, 'richard@udacity.com' => Object {...}}
```

Looping:

1. Using the `MapIterator`

Using both the .keys() and .values() methods on a Map will return a new iterator object called MapIterator. You can store that iterator object in a new variable and use .next() to loop through each key or value. Depending on which method you use, will determine if your iterator has access to the Map's keys or the Map's values.

```
//On Keys
let iteratorObjForKeys = members.keys();
iteratorObjForKeys.next();            //Object {value: 'Evelyn', done: false}
//On values
let iteratorObjForValues = members.values();
iteratorObjForValues.next();          //Object {value: 75.68, done: false}
```

2.Using a for...of Loop

```
/*
 * Using array destructuring, fix the following code to print the keys and values of the `membe
rs` Map to the console.
 */
const members = new Map();

members.set('Evelyn', 75.68);
members.set('Liam', 20.16);
members.set('Sophia', 0);
members.set('Marcus', 10.25);

for (const member of members) {
    let arr = [a,b] = member;         //Member is an array of ['Evelyn',75.68], and with es6 de
structuring, we can access the values
```

```
    console.log(a, b);
}
```

3. Using a forEach Loop

```
members.forEach((key, value) => console.log(key, value));
 //'Evelyn' 75.68
 //'Liam' 20.16
 //'Sophia' 0
 //'Marcus' 10.25
```

WeakMap

A WeakMap is just like a normal Map with a few key differences:

1. a WeakMap can only contain objects as keys,
2. a WeakMap is not iterable which means it can't be looped and
3. a WeakMap does not have a .clear() method.

```
const book1 = { title: 'Pride and Prejudice', author: 'Jane Austen' };
const book2 = { title: 'The Catcher in the Rye', author: 'J.D. Salinger' };
const book3 = { title: 'Gulliver's Travels', author: 'Jonathan Swift' };

const library = new WeakMap();
library.set(book1, true);
library.set(book2, false);
library.set(book3, true);

console.log(library);
WeakMap {Object {title: 'Pride and Prejudice', author: 'Jane Austen'} => true, Object {title:
'The Catcher in the Rye', author: 'J.D. Salinger'} => false, Object {title: 'Gulliver's Travel
s', author: 'Jonathan Swift'} => true}
```

...but if you try to add something other than an object as a key, you'll get an error!

```
library.set('The Grapes of Wrath', false);

//Uncaught TypeError: Invalid value used as weak map key(…)
```

If an object will become null, cease to exists then it will be removed from the WeakMap as well.

```
book1 = null;

console.log(library);

WeakMap {Object {title: 'The Catcher in the Rye', author: 'J.D. Salinger'} => false, Object {ti

tle: 'Gulliver's Travels', author: 'Jonathan Swift'} => true}
```

Promises:

A JavaScript Promise is created with the new Promise constructor function - `new Promise()`. A promise will let you start some work that will 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 provide this code as the argument of the constructor function:

- reject() -->Will notify us if the request hasn't been completed
- resolve() -->Will notify us if the request has been completed or not.

```
new Promise(function (resolve, reject) {

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

        const sundae = {};

        // request ice cream

        // get cone

        // warm up ice cream scoop

        // scoop generous portion into cone!

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

            reject(`Sorry, we're out of that flavor :-(`);

        }

        resolve(sundae);

    }, Math.random() * 2000);

});
```

Promises Return Immediately
The first thing to understand is that a Promise will immediately return an object.

That object has a `.then()` method on it that we can use to have it notify us if the request we made in the promise was either successful or failed. The .then() method takes two functions:
.then(resolve(),reject())

- the function to run if the request completed successfully
- the function to run if the request failed to complete

```
mySundae.then(function(sundae) {
    console.log(`Time to eat my delicious ${sundae}`);
}, function(msg) {
    console.log(msg);
    console.log('crying'); // not a real method
});
```

Proxies

(Objects which handles other object's queries)
To create a proxy object, we use the Proxy constructor - `new Proxy();`. The proxy constructor takes two items:

- new Proxy(object, handler)
- the object that it will be the proxy for
- an object containing the list of methods it will handle for the proxied object

The second object is called the **handler**.

- the handler object is made up of 1 of 13 different "traps"
- a trap is a function that will intercept calls to properties let you run code
- if a trap is not defined, the default behavior is sent to the target object

Get Trap
The get trap is used to "intercept" calls to properties:

```
const richard = {status: 'looking for work'};
const handler = {
    get(target, propName) {
        console.log(target); // the `richard` object, not `handler` and not `agent`
        console.log(propName); // the name of the property the proxy (`agent` in this case) is
 checking
        return target[propName]; Returns the richard.status' text.
    }
};
```

```
//Notice we added the return target[propName]; as the last line of the get trap. This will acce
ss the property on the target object and will return it.
const agent = new Proxy(richard, handler);
agent.status;
```

Having the proxy return info, directly

```
const richard = {status: 'looking for work'};
const handler = {
    get(target, propName) {
        return `He's following many leads, so you should offer a contract as soon as
possible!`;
    }
};
const agent = new Proxy(richard, handler);
agent.status; // returns the text `He's following many leads, so you should offer a contract as
 soon as possible!`
```

Set Trap

The set trap is used for intercepting code that will change a property.
The set trap receives: **(Object,Property,Value)**

```
const richard = {status: 'looking for work'};
const handler = {
    set(target, propName, value) {
        if (propName === 'payRate') { // if the pay is being set, take 15% as commission
            value = value * 0.85;
        }
        target[propName] = value;
    }
};
const agent = new Proxy(richard, handler);
agent.payRate = 1000; // set the actor's pay to $1,000
```

```
agent.payRate; // $850 the actor's actual pay
```

Other types of trap

1. the get trap - lets the proxy handle calls to property access
2. the set trap - lets the proxy handle setting the property to a new value
3. the apply trap - lets the proxy handle being invoked (the object being proxied is a function)
4. the has trap - lets the proxy handle the using in operator
5. the deleteProperty trap - lets the proxy handle if a property is deleted
6. the ownKeys trap - lets the proxy handle when all keys are requested
7. the construct trap - lets the proxy handle when the proxy is used with the new keyword as a constructor
8. the defineProperty trap - lets the proxy handle when defineProperty is used to create a new property on the object
9. the getOwnPropertyDescriptor trap - lets the proxy handle getting the property's descriptors
0. the preventExtenions trap - lets the proxy handle calls to Object.preventExtensions() on the proxy object
1. the isExtensible trap - lets the proxy handle calls to Object.isExtensible on the proxy object
2. the getPrototypeOf trap - lets the proxy handle calls to Object.getPrototypeOf on the proxy object
3. the setPrototypeOf trap - lets the proxy handle calls to Object.setPrototypeOf on the proxy object

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy/handler

Proxy & ES5 Getter/Setter

ES5 Getter/Setter
- We can right our own getter, setter methods for the properties, but when we would like to dynamicall add new propertynames, it won't work. (**weight.**)

```
var obj = {

    _age: 5,

    _height: 4,

    get age() {

        console.log(`getting the "age" property`);

        console.log(this._age);

    },

    get height() {

        console.log(`getting the "height" property`);

        console.log(this._height);

    }

};

obj.age; // logs 'getting the "age" property' & 5
```

```
obj.height; // logs 'getting the "height" property' & 4

But look what happens when we now add a new property to the object:


obj.weight = 120; // set a new property on the object

obj.weight; // logs just 120
```

Proxy:
- Here we can dynamically write get functions with current and future properties.

```
const proxyObj = new Proxy({age: 5, height: 4}, {

    get(targetObj, property) {

        console.log(`getting the ${property} property`);

        console.log(targetObj[property]);

    }

});


proxyObj.age; // logs 'getting the age property' & 5

proxyObj.height; // logs 'getting the height property' & 4

All well and good, just like the ES5 code, but look what happens when we add a new property:


proxyObj.weight = 120; // set a new property on the object

proxyObj.weight; // logs 'getting the weight property' & 120
```

Generators

When a generator is invoked, it doesn't actually run any of the code inside the function. Instead, it creates and returns an iterator. This iterator can then be used to execute the actual generator's inner code.

Pausable Functions

- function* names() { /* ... */ }
- function * names() { /* ... */ }
- function *names() { /* ... */ }

Stopping the function (& returning value)

```
function* getEmployee() {

    console.log('the function has started');
```

```
    const names = ['Amanda', 'Diego', 'Farrin', 'James', 'Kagure', 'Kavita', 'Orit',
'Richard'];
    for (const name of names) {
        console.log(name);
        yield;
        // or return a a input "return yield name"
    }
    console.log('the function has ended');
}
//Notice that there's now a yield inside the for...of loop. If we invoke the generator (which p
roduces an iterator) and then call .next(), we'll get the following output:
const generatorIterator = getEmployee();
generatorIterator.next();   //the function has started    // Amanda
//It's paused! But to really be sure, let's check out the next iteration:
generatorIterator.next();   // Logs out Diego
```

Get data out of a generator by **using** the `yield` keyword. We can also send data back into the generator, too. We do this using the `.next()` method:
(With r**esponse variable = yield**, we are adding data into the generator)

```
function* displayResponse() {
    const response = yield;
    console.log(`Your response is "${response}"!`);
}

const iterator = displayResponse();
iterator.next(); // starts running the generator function
iterator.next('Hello Udacity Student'); // send data into the generator
// the line above logs to the console: Your response is "Hello Udacity Student"!
```

Send Data into the generator
(With **yield variable** we can reference the variable with ${variable}) Thus the names are not modified, but can be used from the array.

```
function* getEmployee() {
    const names = ['Amanda', 'Diego', 'Farrin', 'James', 'Kagure', 'Kavita', 'Orit',
'Richard'];
    const facts = [];
    for (const name of names) {
        // yield *out* each name AND store the returned data into the facts array
        facts.push(yield name);
    }
    return facts;
}
const generatorIterator = getEmployee();

// get the first name out of the generator
let name = generatorIterator.next().value;

// pass data in *and* get the next name
name = generatorIterator.next(`${name} is cool!`).value;
name = generatorIterator.next(`${name} is awesome!`).value;
name = generatorIterator.next(`${name} is stupendous!`).value;
name = generatorIterator.next(`${name} is rad!`).value;
name = generatorIterator.next(`${name} is impressive!`).value;
name = generatorIterator.next(`${name} is stunning!`).value;
name = generatorIterator.next(`${name} is awe-inspiring!`).value;

// pass the last data in, generator ends and returns the array
const positions = generatorIterator.next(`${name} is magnificent!`).value;

// displays each name with description on its own line
positions.join('\n');
```

ES6 Syntax Deploy

What is a polyfill?
A polyfill, or polyfiller, is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively.

Sample

```
if (!String.prototype.startsWith) {
  String.prototype.startsWith = function (searchString, position) {
    position = position || 0;
    return this.substr(position, searchString.length) === searchString;
  };
}
```

As you can see, a polyfill is just regular JavaScript.

Link for al polyfills:
https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills

Complier (Source code ---> Machine Code)
Transpiler (Source code ES6---> Source code ES5)

Tutorial for Babel package (Node Js)
https://www.codementor.io/iykyvic/writing-your-nodejs-apps-using-es6-6dh0edw2o