JavaScript ES6

Lesson 15
Arrays & Collections

Capgemini

Lesson Objectives

At the end of this module you will be able to:

- Perform operations on the new methods introduced in Array
- Implement the iteration protocols in the objects
- Create and use the generator objects

ES6 Library – Arrays

ES6 adds new properties to the global Array object and to its instances to make working with arrays easier.

Array.from(iterable, mapFunc, this) : Creates a new array instance from an iterable object.

- The first argument is a reference to the iterable object.
- The second argument is optional and is a callback (known as Map function) that is called for every element of the iterable object
- The third argument is also optional and is the value of this inside the Map function.

Array.of(values…) : It is an alternative to the Array constructor for creating arrays.

- Array constructor constructs an empty array with array length property equal to the passed number instead of creating an array of one element with that number in it. Array.of() solves this issue.

ES6 Library – Arrays

fill(value, startIndex, endIndex) : Fills all the elements of the array with a given value.

- startIndex and endIndex arguments are optional; if they are not provided then the whole array is filled with the given value
- If only startIndex is provided then endIndex defaults to length of array minus 1.
- If startIndex is negative then it's treated as length of array plus startIndex. If endIndex is negative, it is treated as length of array plus endIndex.

ES6 Library – Arrays

find(testingFunc, this) : Returns an array element, if it satisfies the provided testing function. Otherwise it returns undefined.

- first argument is the testing function and the second argument is the value of this in the testing function. The second argument is optional
- The testing function has three parameters: the first parameter is the array element being processed, second parameter is the index of the current element being processed and third parameter is the array on which find() is called upon.
- If only startIndex is provided then endIndex defaults to length of array minus 1.
- If startIndex is negative then it's treated as length of array plus startIndex. If endIndex is negative, it is treated as length of array plus endIndex.

findIndex(testingFunc, this) : returns the index of the satisfied array element instead of the element itself.

ES6 Library – Arrays

copyWithin(targetIndex, startIndex, endIndex): Used to copy the sequence of values of the array to a different position in the array.

- first argument represents the target index where to copy elements to, second argument represents the index position where to start copying from and the third argument represents the index , that is, where to actually end copying elements
- The third argument is optional. If startIndex is negative then it's calculated as length+startIndex. Similarly if endIndex is negative then it's calculated as length+endIndex.

ES6 Library – Arrays

entries() : returns an iterable object that contains key/value pairs for each index of the array.

keys(): returns an iterable object that contains keys for each of the indexes in the array

values() : returns an iterable object that contains values of the array.

Demo

Array-Extensions

Map

A Map is a collection of key/value pairs.

Keys and values of a Map can be of any data type

The key/value pairs are arranged in the insertion order.

If key already exists, then it's overwritten

The Map objects also implement the iterable protocol and so that it can be used as an iterable object.

A Map object is created using the Map constructor.

Map

```
let map = new Map([
[ 1, 'one' ],
[ 2, 'two' ],
[ 3, 'three' ], // trailing comma is ignored
]);
```

```
let map = new Map()
.set(1, 'one')
.set(2, 'two')
.set(3, 'three');
```

```
let map = new Map();

const KEY1 = {};
map.set(KEY1, 'hello');
console.log(map.get(KEY1)); // hello

const KEY2 = {};
map.set(KEY2, 'world');
console.log(map.get(KEY2)); // world
```

WeakMap

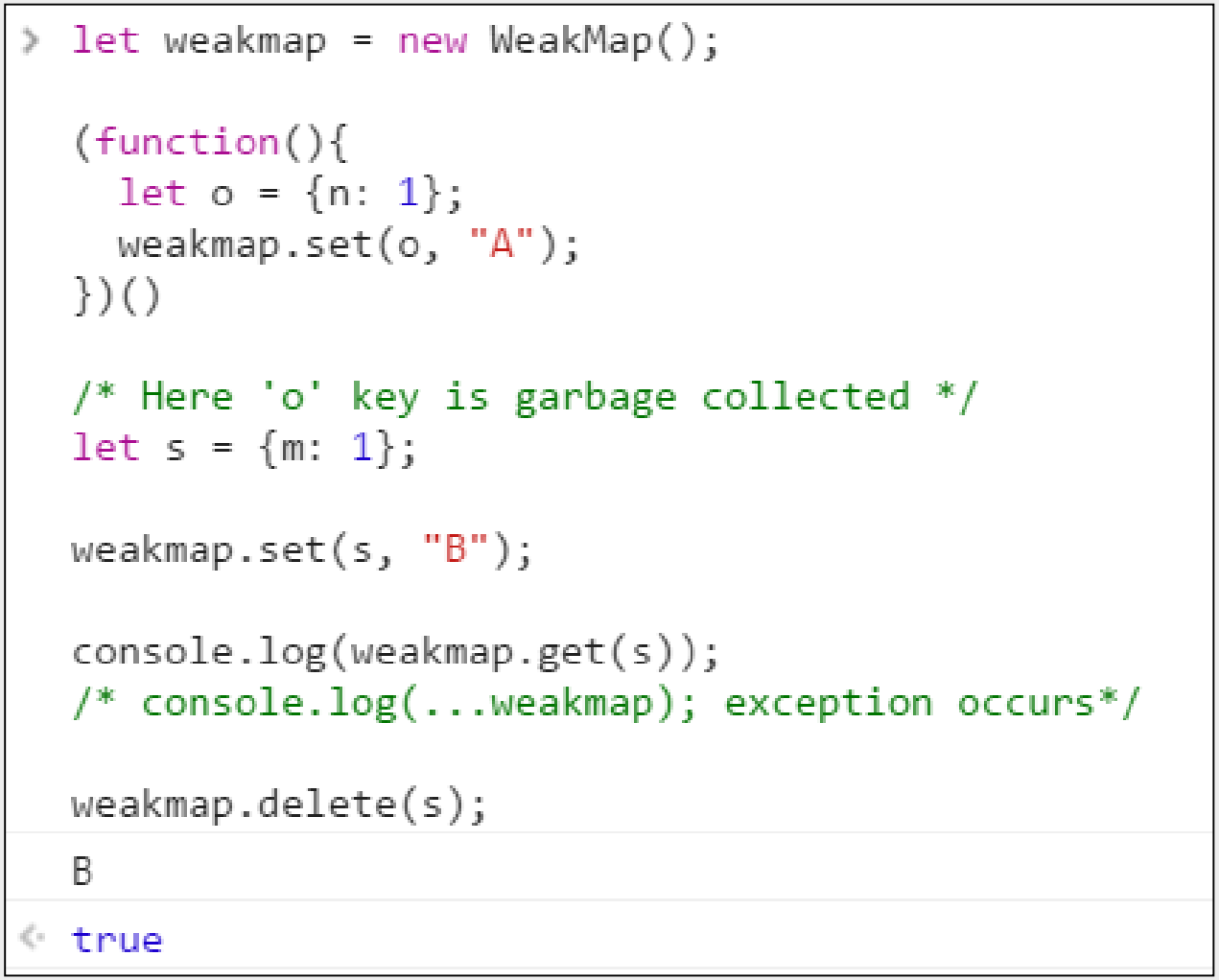
Keys of a Map object can be of primitive types or object references but keys in WeakMap object can only be object references

If there is no other reference to an object i.e. referenced by a key; then the key is garbage collected.

WeakMap object is not enumerable. i.e. size cannot be determined and it doesn't implement iterable protocol.

A WeakMap is created using WeakMap constructor

WeakMap



```
> let weakmap = new WeakMap();

  (function(){
    let o = {n: 1};
    weakmap.set(o, "A");
  })()

  /* Here 'o' key is garbage collected */
  let s = {m: 1};

  weakmap.set(s, "B");

  console.log(weakmap.get(s));
  /* console.log(...weakmap); exception occurs*/

  weakmap.delete(s);
  B
<- true
```

Set

A Set is a collection of unique values of any data type.

The values in a Set are arranged in insertion order.

The Set object implements the iterable protocol so they can be used as an iterable object.

A Set is created using Set constructor.

Iterable object can be passed as an argument to Set object.

Weak Set

A Set can store primitive types and object references whereas a WeakSet object can only store object references

If there is no other reference to an object stored in a WeakSet object then they are garbage collected.

WeakSet object is not enumerable, that is, you cannot find its size; it also doesn't implement iterable protocol.

A WeakSet object is created using WeakSet constructor. You cannot pass an iterable object as an argument to WeakSet object.

WeakSet

```
> let weakset = new WeakSet();

  (function(){
    let a = {};
    weakset.add(a);
  })()

  //here 'a' is garbage collected from weakset
  console.log(weakset.size); //output "undefined"
  console.log(...weakset); //Exception is thrown

  weakset.clear(); //Exception, no such function
```

iterators in ES6

In ES6 arrays have a special property (symbol) Symbol.iterator.

```
> var numbers = [1,44,55];
> var iterator = numbers[Symbol.iterator]();
> iterator.next().value;
< 1
> iterator.next().value;
< 44
> iterator.next().value;
< 55
> iterator.next();
< Object {value: undefined, done: true}
```

Alternate way to iterate array in ES6

```
> var numbers = [1,44,55];
> for(var number of numbers){
    console.log(number);
  }
  1
  44
  55
```

In ES6 iterators are built into Arrays, String, Map, Set by default and custom iterators can be created for Object

iteration protocols

An iteration protocol is a set of rules that an object needs to follow for implementing the interface, which when used, a loop or a construct can iterate over a group of values of the object.

ES6 introduces two new iteration protocols

- **Iterator protocol** : Any object that implements the iterator protocol is known as an iterator. According to the iterator protocol, an object needs to provide a next() method that returns the next item in the sequence of a group of items
- **iterable protocol**: Any object that implements the iterable protocol is known as an iterable. According to the iterable protocol, an object needs to provide the @@iterator method; i.e. it must have the Symbol.iterator symbol as a property key.
- The @@iterator method must return an iterator object.

Implementing Iteration Protocols

Object Implementing iterator protocol

```
> let obj = {
    array: [1, 2],
    nextIndex: 0,
    next: function(){
      return this.nextIndex < this.array.length ?
      {value: this.array[this.nextIndex++], done: false} :
      {done: true};
    }
  };

> obj.next()
< Object {value: 1, done: false}
> obj.next()
< Object {value: 2, done: false}
> obj.next()
< Object {done: true}
```

Object Implementing iterable protocol

```
> let obj = {
    array: [1, 2],
    nextIndex: 0,
    [Symbol.iterator]: function(){
      return {
        array: this.array,
        nextIndex: this.nextIndex,
        next: function(){
          return this.nextIndex < this.array.length ?
          {value: this.array[this.nextIndex++], done: false} :
          {done: true};
        }
      }
    }
  };

> let iterable = obj[Symbol.iterator]();

> iterable.next();
< Object {value: 1, done: false}
> iterable.next();
< Object {value: 2, done: false}
> iterable.next();
< Object {done: true}
```

Demo

iterator

Generators

A generator function is like a normal function, but instead of returning a single value, it returns multiple values one by one.

A generator function is written using the function* expression.

Generator object implements both iterable and iterator protocols.

Every generator object holds a new execution context of the generator function. When the next() method of the generator object called, it executes the generator function's body until the yield keyword is encountered. It returns the yielded value, and pauses the function. When the next() method is called again, it resumes the execution, and then returns the next yielded value.

The done property is true when the generator function doesn't yield any more value.

Yielding and throwing exceptions in generators

Generator can yield the value and the value will be returned by calling next() of generator object.

If value for yield is omitted then undefined is returned.

We can also pass an optional argument to the next() method. This argument becomes the value returned by the yield statement, where the generator function is currently paused.

The yield* keyword inside a generator function takes an iterable object as the expression and iterates it to yield its values

Exceptions can be manually triggered inside a generator function using the throw() method of the generator object.

Demo

generators

Summary

ES6 adds new properties to the global Array object and to its instances to make working with arrays easier.

A collection is any object that stores multiple elements as a single unit.

In ES6 iterators are built into Arrays, String, Map, Set by default and custom iterators can be created for Object

for…of loop is used to iterate over an iterable object.

The @@iterator method must return an iterator object.

A generator function is like a normal function, but instead of returning a single value, it returns multiple values one by one.