Chapter 76: WeakMap

Section 76.1: Creating a WeakMap object

WeakMap object allows you to store key/value pairs. The difference from Map is that keys must be objects and are weakly referenced. This means that if there aren't any other strong references to the key, the element in WeakMap can be removed by garbage collector.

WeakMap constructor has an optional parameter, which can be any iterable object (for example Array) containing key/value pairs as two-element arrays.

```
const o1 = {a: 1, b: 2},
      o2 = {};

const weakmap = new WeakMap([[o1, true], [o2, o1]]);
```

Section 76.2: Getting a value associated to the key

To get a value associated to the key, use the `.get()` method. If there's no value associated to the key, it returns `undefined`.

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.get(obj1)); // 7
console.log(weakmap.get(obj2)); // undefined
```

Section 76.3: Assigning a value to the key

To assign a value to the key, use the `.set()` method. It returns the WeakMap object, so you can chain `.set()` calls.

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap();
weakmap.set(obj1, 1).set(obj2, 2);
console.log(weakmap.get(obj1)); // 1
console.log(weakmap.get(obj2)); // 2
```

Section 76.4: Checking if an element with the key exists

To check if an element with a specified key exits in a WeakMap, use the `.has()` method. It returns `true` if it exits, and otherwise `false`.

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.has(obj1)); // true
console.log(weakmap.has(obj2)); // false
```

Section 76.5: Removing an element with the key

To remove an element with a specified key, use the `.delete()` method. It returns `true` if the element existed and has been removed, otherwise `false`.

```
const obj1 = {},
      obj2 = {};

const weakmap = new WeakMap([[obj1, 7]]);
console.log(weakmap.delete(obj1)); // true
console.log(weakmap.has(obj1)); // false
console.log(weakmap.delete(obj2)); // false
```

Section 76.6: Weak reference demo

JavaScript uses reference counting technique to detect unused objects. When reference count to an object is zero, that object will be released by the garbage collector. Weakmap uses weak reference that does not contribute to reference count of an object, therefore it is very useful to solve memory leak problems.

Here is a demo of weakmap. I use a very large object as value to show that weak reference does not contribute to reference count.

```
// manually trigger garbage collection to make sure that we are in good status.
> global.gc();
undefined

// check initial memory use , heapUsed is 4M or so
> process.memoryUsage();
{ rss: 21106688,
  heapTotal: 7376896,
  heapUsed: 4153936,
  external: 9059 }

> let wm = new WeakMap();
undefined

> const b = new Object();
undefined

> global.gc();
undefined

// heapUsed is still 4M or so
> process.memoryUsage();
{ rss: 20537344,
  heapTotal: 9474048,
  heapUsed: 3967272,
  external: 8993 }

// add key-value tuple into WeakMap ,
// key is b , value is 5*1024*1024 array
> wm.set(b, new Array(5*1024*1024));
WeakMap {}

// manually garbage collection
> global.gc();
undefined

// heapUsed is still 45M
```

```
> process.memoryUsage();
{ rss: 62652416,
  heapTotal: 51437568,
  heapUsed: 45911664,
  external: 8951 }

// b reference to null
> b = null;
null

// garbage collection
> global.gc();
undefined

// after remove b reference to object , heapUsed is 4M again
// it means the big array in WeakMap is released
// it also means weekmap does not contribute to big array's reference count, only b does.
> process.memoryUsage();
{ rss: 20639744,
  heapTotal: 8425472,
  heapUsed: 3979792,
  external: 8956 }
```