Sets and Maps

JavaScript only had one type of collection, represented by the `Array` type, for most of its history (though some may argue all non-array objects are just collections of key-value pairs, their intended use was, originally quite different from arrays). Arrays are used in JavaScript just like arrays in other languages, but the lack of other collection options meant arrays were often used as queues and stacks, as well. Since arrays only use numeric indices, developers used non-array objects whenever a non-numeric index was necessary. That technique led to custom implementations of sets and maps using non-array objects.

A *set* is a list of values that cannot contain duplicates. You typically don't access individual items in a set like you would items in an array; instead, it's much more common to just check a set to see if a value is present. A *map* is a collection of keys that correspond to specific values. As such, each item in a map stores two pieces of data, and values are retrieved by specifying the key to read from. Maps are frequently used as caches, for storing data to be quickly retrieved later. While ECMAScript 5 didn't formally have sets and maps, developers worked around this limitation using non-array objects, too.

ECMAScript 6 added sets and maps to JavaScript, and this chapter discusses everything you need to know about these two collection types.

First, I will discuss the workarounds developers used to implement sets and maps before ECMAScript 6, and why those implementations were problematic. After that important background information, I will cover how sets and maps work in ECMAScript 6.

Sets and Maps in ECMAScript 5

In ECMAScript 5, developers mimicked sets and maps by using object properties, like this:

```
let set = Object.create(null);

set.foo = true;

// checking for existence
if (set.foo) {

    // do something
}
```

The `set` variable in this example is an object with a `null` prototype, ensuring that there are no inherited properties on the object. Using object properties as unique values to be checked is a common approach in ECMAScript 5. When a property is added to the `set` object, it is set to `true` so conditional statements (such as the `if` statement in this example) can easily check whether the value is present.

The only real difference between an object used as a set and an object used as a map is the value being stored. For instance, this example uses an object as a map:

```
let map = Object.create(null);

map.foo = "bar";

// retrieving a value
let value = map.foo;

console.log(value);         // "bar"
```

This code stores a string value `"bar"` under the key `foo`. Unlike sets, maps are mostly used to retrieve information, rather than just checking for the key's existence.

Problems with Workarounds

While using objects as sets and maps works okay in simple situations, the approach can get more complicated once you run into the limitations of object properties. For example, since all object properties must be strings, you must be certain no two keys evaluate to the same string. Consider the following:

```
let map = Object.create(null);

map[5] = "foo";

console.log(map["5"]);      // "foo"
```

This example assigns the string value `"foo"` to a numeric key of `5`. Internally, that numeric value is converted to a string, so `map["5"]` and `map[5]` actually reference the same property. That internal conversion can cause problems when you want to use both numbers and strings as keys. Another problem arises when using objects as keys, like this:

```
let map = Object.create(null),
    key1 = {},
    key2 = {};

map[key1] = "foo";

console.log(map[key2]);     // "foo"
```

Here, `map[key2]` and `map[key1]` reference the same value. The objects `key1` and `key2` are converted to strings because object properties must be strings. Since `"[object Object]"` is the default string representation for objects, both `key1` and `key2` are converted to that string. This can cause errors that may not be obvious because it's logical to assume that different object keys would, in fact, be different.

The conversion to the default string representation makes it difficult to use objects as keys. (The same problem exists when trying to use an object as a set.)

Maps with a key whose value is falsy present their own particular problem, too. A falsy value is automatically converted to false when used in situations where a boolean value is required, such as in the condition of an `if` statement. This conversion alone isn't a problem--so long as you're careful as to how you use values. For instance, look at this code:

```
let map = Object.create(null);

map.count = 1;

// checking for the existence of "count" or a nonzero value?
if (map.count) {
    // ...
}
```

This example has some ambiguity as to how `map.count` should be used. Is the `if` statement intended to check for the existence of `map.count` or that the value is nonzero? The code inside the `if` statement will execute because the value 1 is truthy. However, if `map.count` is 0, or if `map.count` doesn't exist, the code inside the `if` statement would not be executed.

These are difficult problems to identify and debug when they occur in large applications, which is a prime reason that ECMAScript 6 adds both sets and maps to the language.

I> JavaScript has the `in` operator that returns `true` if a property exists in an object without reading the value of the object. However, the `in` operator also searches the prototype of an object, which makes it only safe to use when an object has a `null` prototype. Even so, many developers still incorrectly use code as in the last example rather than using `in`.

Sets in ECMAScript 6

ECMAScript 6 adds a `Set` type that is an ordered list of values without duplicates. Sets allow fast access to the data they contain, adding a more efficient manner of tracking discrete values.

Creating Sets and Adding Items

Sets are created using `new Set()` and items are added to a set by calling the `add()` method. You can see how many items are in a set by checking the `size` property:

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.size);    // 2
```

Sets do not coerce values to determine whether they are the same. That means a set can contain both the number `5` and the string `"5"` as two separate items. (The only exception is that -0 and +0 are considered to be the same.) You can also add multiple objects to the set, and those objects will remain distinct:

```
let set = new Set(),
    key1 = {},
    key2 = {};

set.add(key1);
set.add(key2);

console.log(set.size);    // 2
```

Because key1 and key2 are not converted to strings, they count as two unique items in the set. (Remember, if they were converted to strings, they would both be equal to "[object Object]".)

If the add() method is called more than once with the same value, all calls after the first one are effectively ignored:

```
let set = new Set();
set.add(5);
set.add("5");
set.add(5);     // duplicate - this is ignored

console.log(set.size);    // 2
```

You can initialize a set using an array, and the Set constructor will ensure that only unique values are used. For instance:

```
let set = new Set([1, 2, 3, 4, 5, 5, 5, 5]);
console.log(set.size);    // 5
```

In this example, an array with duplicate values is used to initialize the set. The number 5 only appears once in the set even though it appears four times in the array. This functionality makes converting existing code or JSON structures to use sets easy.

I> The Set constructor actually accepts any iterable object as an argument. Arrays work because they are iterable by default, as are sets and maps. The Set constructor uses an iterator to extract values from the argument. (Iterables and iterators are discussed in detail in Chapter 8.)

You can test which values are in a set using the has() method, like this:

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.has(5));    // true
console.log(set.has(6));    // false
```

Here, `set.has(6)` would return false because the set doesn't have that value.

Removing Values

It's also possible to remove values from a set. You can remove single value by using the `delete()` method, or you can remove all values from the set by calling the `clear()` method. This code shows both in action:

```
let set = new Set();
set.add(5);
set.add("5");

console.log(set.has(5));    // true

set.delete(5);

console.log(set.has(5));    // false
console.log(set.size);      // 1

set.clear();

console.log(set.has("5"));  // false
console.log(set.size);      // 0
```

After the `delete()` call, only `5` is gone; after the `clear()` method executes, `set` is empty.

All of this amounts to a very easy mechanism for tracking unique ordered values. However, what if you want to add items to a set and then perform some operation on each item? That's where the `forEach()` method comes in.

The forEach() Method for Sets

If you're used to working with arrays, then you may already be familiar with the `forEach()` method. ECMAScript 5 added `forEach()` to arrays to make working on each item in an array without setting up a `for` loop easier. The method proved popular among developers, and so the same method is available on sets and works the same way.

The `forEach()` method is passed a callback function that accepts three arguments:

1. The value from the next position in the set
2. The same value as the first argument
3. The set from which the value is read

The strange difference between the set version of `forEach()` and the array version is that the first and second arguments to the callback function are the same. While this might look like a mistake, there's a good reason for the behavior.

The other objects that have `forEach()` methods (arrays and maps) pass three arguments to their callback functions. The first two arguments for arrays and maps are the value and the key (the numeric index for arrays).

Sets do not have keys, however. The people behind the ECMAScript 6 standard could have made the callback function in the set version of `forEach()` accept two arguments, but that would have made it different from the other two. Instead, they found a way to keep the callback function the same and accept three arguments: each value in a set is considered to be both the key and the value. As such, the first and second argument are always the same in `forEach()` on sets to keep this functionality consistent with the other `forEach()` methods on arrays and maps.

Other than the difference in arguments, using `forEach()` is basically the same for a set as it is for an array. Here's some code that shows the method at work:

```
let set = new Set([1, 2]);

set.forEach(function(value, key, ownerSet) {
    console.log(key + " " + value);
    console.log(ownerSet === set);
});
```

This code iterates over each item in the set and outputs the values passed to the `forEach()` callback function. Each time the callback function executes, `key` and `value` are the same, and `ownerSet` is always equal to `set`. This code outputs:

```
1 1
true
2 2
true
```

Also the same as arrays, you can pass a `this` value as the second argument to `forEach()` if you need to use `this` in your callback function:

```
let set = new Set([1, 2]);

let processor = {
    output(value) {
        console.log(value);
    },
    process(dataSet) {
        dataSet.forEach(function(value) {
            this.output(value);
        }, this);
    }
};

processor.process(set);
```

In this example, the `processor.process()` method calls `forEach()` on the set and passes `this` as the `this` value for the callback. That's necessary so `this.output()` will correctly resolve to the `processor.output()` method. The `forEach()` callback function only makes use of the first argument, `value`, so the others are omitted. You can also use an arrow function to get the same effect without passing the second argument, like this:

```
let set = new Set([1, 2]);

let processor = {
    output(value) {
        console.log(value);
    },
    process(dataSet) {
        dataSet.forEach((value) => this.output(value));
    }
};

processor.process(set);
```

The arrow function in this example reads `this` from the containing `process()` function, and so it should correctly resolve `this.output()` to a `processor.output()` call.

Keep in mind that while sets are great for tracking values and `forEach()` lets you work on each value sequentially, you can't directly access a value by index like you can with an array. If you need to do so, then the best option is to convert the set into an array.

Converting a Set to an Array

It's easy to convert an array into a set because you can pass the array to the `Set` constructor. It's also easy to convert a set back into an array using the spread operator. Chapter 3 introduced the spread operator (`...`) as a way to split items in an array into separate function parameters. You can also use the spread operator to work on iterable objects, such as sets, to convert them into arrays. For example:

```
let set = new Set([1, 2, 3, 3, 3, 4, 5]),
    array = [...set];

console.log(array);             // [1,2,3,4,5]
```

Here, a set is initially loaded with an array that contains duplicates. The set removes the duplicates, and then the items are placed into a new array using the spread operator. The set itself still contains the same items (`1`, `2`, `3`, `4`, and `5`) it received when it was created. They've just been copied to a new array.

This approach is useful when you already have an array and want to create an array without duplicates. For example:

```
function eliminateDuplicates(items) {
    return [...new Set(items)];
}
```

```
let numbers = [1, 2, 3, 3, 3, 4, 5],
    noDuplicates = eliminateDuplicates(numbers);

console.log(noDuplicates);      // [1,2,3,4,5]
```

In the `eliminateDuplicates()` function, the set is just a temporary intermediary used to filter out duplicate values before creating a new array that has no duplicates.

Weak Sets

The `Set` type could alternately be called a strong set, because of the way it stores object references. An object stored in an instance of `Set` is effectively the same as storing that object inside a variable. As long as a reference to that `Set` instance exists, the object cannot be garbage collected to free memory. For example:

```
let set = new Set(),
    key = {};

set.add(key);
console.log(set.size);      // 1

// eliminate original reference
key = null;

console.log(set.size);      // 1

// get the original reference back
key = [...set][0];
```

In this example, setting `key` to `null` clears one reference of the `key` object, but another remains inside `set`. You can still retrieve `key` by converting the set to an array with the spread operator and accessing the first item. That result is fine for most programs, but sometimes, it's better for references in a set to disappear when all other references disappear. For instance, if your JavaScript code is running in a web page and wants to keep track of DOM elements that might be removed by another script, you don't want your code holding onto the last reference to a DOM element. (That situation is called a *memory leak*.)

To alleviate such issues, ECMAScript 6 also includes *weak sets*, which only store weak object references and cannot store primitive values. A *weak reference* to an object does not prevent garbage collection if it is the only remaining reference.

Creating a Weak Set

Weak sets are created using the `WeakSet` constructor and have an `add()` method, a `has()` method, and a `delete()` method. Here's an example that uses all three:

```
let set = new WeakSet(),
    key = {};
```

```
// add the object to the set
set.add(key);

console.log(set.has(key));      // true

set.delete(key);

console.log(set.has(key));      // false
```

Using a weak set is a lot like using a regular set. You can add, remove, and check for references in the weak set. You can also seed a weak set with values by passing an iterable to the constructor:

```
let key1 = {},
    key2 = {},
    set = new WeakSet([key1, key2]);

console.log(set.has(key1));     // true
console.log(set.has(key2));     // true
```

In this example, an array is passed to the `WeakSet` constructor. Since this array contains two objects, those objects are added into the weak set. Keep in mind that an error will be thrown if the array contains any non-object values, since `WeakSet` can't accept primitive values.

Key Differences Between Set Types

The biggest difference between weak sets and regular sets is the weak reference held to the object value. Here's an example that demonstrates that difference:

```
let set = new WeakSet(),
    key = {};

// add the object to the set
set.add(key);

console.log(set.has(key));      // true

// remove the last strong reference to key, also removes from weak set
key = null;
```

After this code executes, the reference to `key` in the weak set is no longer accessible. It is not possible to verify its removal because you would need one reference to that object to pass to the `has()` method. This can make testing weak sets a little confusing, but you can trust that the reference has been properly removed by the JavaScript engine.

These examples show that weak sets share some characteristics with regular sets, but there are some key differences. Those are:

1. In a `WeakSet` instance, the `add()` method throws an error when passed a non-object (`has()` and `delete()` always return `false` for non-object arguments).
2. Weak sets are not iterables and therefore cannot be used in a `for-of` loop.
3. Weak sets do not expose any iterators (such as the `keys()` and `values()` methods), so there is no way to programmatically determine the contents of a weak set.
4. Weak sets do not have a `forEach()` method.
5. Weak sets do not have a `size` property.

The seemingly limited functionality of weak sets is necessary in order to properly handle memory. In general, if you only need to track object references, then you should use a weak set instead of a regular set.

Sets give you a new way to handle lists of values, but they aren't useful when you need to associate additional information with those values. That's why ECMAScript 6 also adds maps.

Maps in ECMAScript 6

The ECMAScript 6 `Map` type is an ordered list of key-value pairs, where both the key and the value can have any type. Keys equivalence is determined by using the same approach as `Set` objects, so you can have both a key of `5` and a key of `"5"` because they are different types. This is quite different from using object properties as keys, as object properties always coerce values into strings.

You can add items to maps by calling the `set()` method and passing it a key and the value to associate with the key. You can later retrieve a value by passing the key to the `get()` method. For example:

```
let map = new Map();
map.set("title", "Understanding ES6");
map.set("year", 2016);

console.log(map.get("title"));      // "Understanding ES6"
console.log(map.get("year"));       // 2016
```

In this example, two key-value pairs are stored. The `"title"` key stores a string while the `"year"` key stores a number. The `get()` method is called later to retrieve the values for both keys. If either key didn't exist in the map, then `get()` would have returned the special value `undefined` instead of a value.

You can also use objects as keys, which isn't possible when using object properties to create a map in the old workaround approach. Here's an example:

```
let map = new Map(),
    key1 = {},
    key2 = {};

map.set(key1, 5);
map.set(key2, 42);

console.log(map.get(key1));         // 5
console.log(map.get(key2));         // 42
```

This code uses the objects `key1` and `key2` as keys in the map to store two different values. Because these keys are not coerced into another form, each object is considered unique. This allows you to associate additional data with an object without modifying the object itself.

Map Methods

Maps share several methods with sets. That is intentional, and it allows you to interact with maps and sets in similar ways. These three methods are available on both maps and sets:

- `has(key)` - Determines if the given key exists in the map
- `delete(key)` - Removes the key and its associated value from the map
- `clear()` - Removes all keys and values from the map

Maps also have a `size` property that indicates how many key-value pairs it contains. This code uses all three methods and `size` in different ways:

```
let map = new Map();
map.set("name", "Nicholas");
map.set("age", 25);

console.log(map.size);          // 2

console.log(map.has("name"));   // true
console.log(map.get("name"));   // "Nicholas"

console.log(map.has("age"));    // true
console.log(map.get("age"));    // 25

map.delete("name");
console.log(map.has("name"));   // false
console.log(map.get("name"));   // undefined
console.log(map.size);          // 1

map.clear();
console.log(map.has("name"));   // false
console.log(map.get("name"));   // undefined
console.log(map.has("age"));    // false
console.log(map.get("age"));    // undefined
console.log(map.size);          // 0
```

As with sets, the `size` property always contains the number of key-value pairs in the map. The `Map` instance in this example starts with the `"name"` and `"age"` keys, so `has()` returns `true` when passed either key. After the `"name"` key is removed by the `delete()` method, the `has()` method returns `false` when passed `"name"` and the `size` property indicates one less item. The `clear()` method then removes the remaining key, as indicated by `has()` returning `false` for both keys and `size` being 0.

The `clear()` method is a fast way to remove a lot of data from a map, but there's also a way to add a lot of data to a map at one time.

Map Initialization

Also similar to sets, you can initialize a map with data by passing an array to the `Map` constructor. Each item in the array must itself be an array where the first item is the key and the second is that key's corresponding value. The entire map, therefore, is an array of these two-item arrays, for example:

```
let map = new Map([["name", "Nicholas"], ["age", 25]]);

console.log(map.has("name"));   // true
console.log(map.get("name"));   // "Nicholas"
console.log(map.has("age"));    // true
console.log(map.get("age"));    // 25
console.log(map.size);          // 2
```

The keys `"name"` and `"age"` are added into `map` through initialization in the constructor. While the array of arrays may look a bit strange, it's necessary to accurately represent keys, as keys can be any data type. Storing the keys in an array is the only way to ensure they aren't coerced into another data type before being stored in the map.

The forEach Method on Maps

The `forEach()` method for maps is similar to `forEach()` for sets and arrays, in that it accepts a callback function that receives three arguments:

1. The value from the next position in the map
2. The key for that value
3. The map from which the value is read

These callback arguments more closely match the `forEach()` behavior in arrays, where the first argument is the value and the second is the key (corresponding to a numeric index in arrays). Here's an example:

```
let map = new Map([ ["name", "Nicholas"], ["age", 25]]);

map.forEach(function(value, key, ownerMap) {
    console.log(key + " " + value);
    console.log(ownerMap === map);
});
```

The `forEach()` callback function outputs the information that is passed to it. The `value` and `key` are output directly, and `ownerMap` is compared to `map` to show that the values are equivalent. This outputs:

```
name Nicholas
true
age 25
true
```

The callback passed to `forEach()` receives each key-value pair in the order in which the pairs were inserted into the map. This behavior differs slightly from calling `forEach()` on arrays, where the callback receives each item in order of numeric index.

I> You can also provide a second argument to `forEach()` to specify the `this` value inside the callback function. A call like that behaves the same as the set version of the `forEach()` method.

Weak Maps

Weak maps are to maps what weak sets are to sets: they're a way to store weak object references. In *weak maps*, every key must be an object (an error is thrown if you try to use a non-object key), and those object references are held weakly so they don't interfere with garbage collection. When there are no references to a weak map key outside a weak map, the key-value pair is removed from the weak map.

The most useful place to employ weak maps is when creating an object related to a particular DOM element in a web page. For example, some JavaScript libraries for web pages maintain one custom object for every DOM element referenced in the library, and that mapping is stored in a cache of objects internally.

The difficult part of this approach is determining when a DOM element no longer exists in the web page, so that the library can remove its associated object. Otherwise, the library would hold onto the DOM element reference past the reference's usefulness and cause a memory leak. Tracking the DOM elements with a weak map would still allow the library to associate a custom object with every DOM element, and it could automatically destroy any object in the map when that object's DOM element no longer exists.

I> It's important to note that only weak map keys, and not weak map values, are weak references. An object stored as a weak map value will prevent garbage collection if all other references are removed.

Using Weak Maps

The ECMAScript 6 `WeakMap` type is an unordered list of key-value pairs, where a key must be a non-null object and a value can be of any type. The interface for `WeakMap` is very similar to that of `Map` in that `set()` and `get()` are used to add and retrieve data, respectively:

```
let map = new WeakMap(),
    element = document.querySelector(".element");

map.set(element, "Original");

let value = map.get(element);
console.log(value);             // "Original"

// remove the element
element.parentNode.removeChild(element);
element = null;

// the weak map is empty at this point
```

In this example, one key-value pair is stored. The `element` key is a DOM element used to store a corresponding string value. That value is then retrieved by passing in the DOM element to the `get()` method.

When the DOM element is later removed from the document and the variable referencing it is set to `null`, the data is also removed from the weak map.

Similar to weak sets, there is no way to verify that a weak map is empty, because it doesn't have a `size` property. Because there are no remaining references to the key, you can't retrieve the value by calling the `get()` method, either. The weak map has cut off access to the value for that key, and when the garbage collector runs, the memory occupied by the value will be freed.

Weak Map Initialization

To initialize a weak map, pass an array of arrays to the `WeakMap` constructor. Just like initializing a regular map, each array inside the containing array should have two items, where the first item is the non-null object key and the second item is the value (any data type). For example:

```
let key1 = {},
    key2 = {},
    map = new WeakMap([[key1, "Hello"], [key2, 42]]);

console.log(map.has(key1));     // true
console.log(map.get(key1));     // "Hello"
console.log(map.has(key2));     // true
console.log(map.get(key2));     // 42
```

The objects `key1` and `key2` are used as keys in the weak map, and the `get()` and `has()` methods can access them. An error is thrown if the `WeakMap` constructor receives a non-object key in any of the key-value pairs.

Weak Map Methods

Weak maps have only two additional methods available to interact with key-value pairs. There is a `has()` method to determine if a given key exists in the map and a `delete()` method to remove a specific key-value pair. There is no `clear()` method because that would require enumerating keys, and like weak sets, that isn't possible with weak maps. This example uses both the `has()` and `delete()` methods:

```
let map = new WeakMap(),
    element = document.querySelector(".element");

map.set(element, "Original");

console.log(map.has(element));   // true
console.log(map.get(element));   // "Original"

map.delete(element);
console.log(map.has(element));   // false
console.log(map.get(element));   // undefined
```

Here, a DOM element is once again used as the key in a weak map. The `has()` method is useful for checking to see if a reference is currently being used as a key in the weak map. Keep in mind that this only works when you have a non-null reference to a key. The key is forcibly removed from the weak map by the `delete()` method, at which point `has()` returns `false` and `get()` returns `undefined`.

Private Object Data

While most developers consider the main use case of weak maps to be associated data with DOM elements, there are many other possible uses (and no doubt, some that have yet to be discovered). One practical use of weak maps is to store data that is private to object instances. All object properties are public in ECMAScript 6, and so you need to use some creativity to make data accessible to objects, but not accessible to everything. Consider the following example:

```
function Person(name) {
    this._name = name;
}

Person.prototype.getName = function() {
    return this._name;
};
```

This code uses the common convention of a leading underscore to indicate that a property is considered private and should not be modified outside the object instance. The intent is to use `getName()` to read `this._name` and not allow the `_name` value to change. However, there is nothing standing in the way of someone writing to the `_name` property, so it can be overwritten either intentionally or accidentally.

In ECMAScript 5, it's possible to get close to having truly private data, by creating an object using a pattern such as this:

```
var Person = (function() {

    var privateData = {},
        privateId = 0;

    function Person(name) {
        Object.defineProperty(this, "_id", { value: privateId++ });

        privateData[this._id] = {
            name: name
        };
    }

    Person.prototype.getName = function() {
        return privateData[this._id].name;
    };

    return Person;
}());
```

This example wraps the definition of `Person` in an IIFE that contains two private variables, `privateData` and `privateId`. The `privateData` object stores private information for each instance while `privateId` is used to generate a unique ID for each instance. When the `Person` constructor is called, a nonenumerable, nonconfigurable, and nonwritable `_id` property is added.

Then, an entry is made into the `privateData` object that corresponds to the ID for the object instance; that's where the `name` is stored. Later, in the `getName()` function, the name can be retrieved by using `this._id` as the key into `privateData`. Because `privateData` is not accessible outside of the IIFE, the actual data is safe, even though `this._id` is exposed publicly.

The big problem with this approach is that the data in `privateData` never disappears because there is no way to know when an object instance is destroyed; the `privateData` object will always contain extra data. This problem can be solved by using a weak map instead, as follows:

```
let Person = (function() {

    let privateData = new WeakMap();

    function Person(name) {
        privateData.set(this, { name: name });
    }

    Person.prototype.getName = function() {
        return privateData.get(this).name;
    };

    return Person;
}());
```

This version of the `Person` example uses a weak map for the private data instead of an object. Because the `Person` object instance itself can be used as a key, there's no need to keep track of a separate ID. When the `Person` constructor is called, a new entry is made into the weak map with a key of `this` and a value of an object containing private information. In this case, that value is an object containing only `name`. The `getName()` function retrieves that private information by passing `this` to the `privateData.get()` method, which fetches the value object and accesses the `name` property. This technique keeps the private information private, and destroys that information whenever an object instance associated with it is destroyed.

Weak Map Uses and Limitations

When deciding whether to use a weak map or a regular map, the primary decision to consider is whether you want to use only object keys. Anytime you're going to use only object keys, then the best choice is a weak map. That will allow you to optimize memory usage and avoid memory leaks by ensuring that extra data isn't kept around after it's no longer accessible.

Keep in mind that weak maps give you very little visibility into their contents, so you can't use the `forEach()` method, the `size` property, or the `clear()` method to manage the items. If you need some inspection capabilities, then regular maps are a better choice. Just be sure to keep an eye on memory usage.

Of course, if you only want to use non-object keys, then regular maps are your only choice.

Summary

ECMAScript 6 formally introduces sets and maps into JavaScript. Prior to this, developers frequently used objects to mimic both sets and maps, often running into problems due to the limitations associated with object properties.

Sets are ordered lists of unique values. Values are not coerced to determine equivalence. Sets automatically remove duplicate values, so you can use a set to filter an array for duplicates and return the result. Sets aren't subclasses of arrays, so you cannot randomly access a set's values. Instead, you can use the `has()` method to determine if a value is contained in the set and the `size` property to inspect the number of values in the set. The `Set` type also has a `forEach()` method to process each set value.

Weak sets are special sets that can contain only objects. The objects are stored with weak references, meaning that an item in a weak set will not block garbage collection if that item is the only remaining reference to an object. Weak set contents can't be inspected due to the complexities of memory management, so it's best to use weak sets only for tracking objects that need to be grouped together.

Maps are ordered key-value pairs where the key can be any data type. Similar to sets, keys are not coerced to determine equivalence, which means you can have a numeric key `5` and a string `"5"` as two separate keys. A value of any data type can be associated with a key using the `set()` method, and that value can later be retrieved by using the `get()` method. Maps also have a `size` property and a `forEach()` method to allow for easier item access.

Weak maps are a special type of map that can only have object keys. As with weak sets, an object key reference is weak and doesn't prevent garbage collection when it's the only remaining reference to an object. When a key is garbage collected, the value associated with the key is also removed from the weak map. This memory management aspect makes weak maps uniquely suited for correlating additional information with objects whose lifecycles are managed outside of the code accessing them.