CHAPTER 9
Sets

A `set` is a collection of unique elements. The elements of a set are called members. The two most important properties of sets are that the members of a set are unordered and that no member can occur in a set more than once. Sets play a very important role in computer science but are not considered a data type in many programming languages. Sets can be useful when you want to create a data structure that consists only of unique elements, such as a list of each unique word in a text. This chapter discusses how to create a `Set` class for JavaScript.

Fundamental Set Definitions, Operations, and Properties

A set is an unordered collection of related members in which no member occurs more than once. A set is denoted mathematically as a list of members surrounded by curly braces, such as {0,1,2,3,4,5,6,7,8,9}. We can write a set in any order, so the previous set can be written as {9,0,8,1,7,2,6,3,5,4} or any other combination of the members such that all the members are written just once.

Set Definitions

Here are some definitions you need to know to work with sets:

- A set containing no members is called the *empty set*. The *universe* is the set of all possible members.
- Two sets are considered equal if they contain exactly the same members.
- A set is considered a *subset* of another set if all the members of the first set are contained in the second set.

Set Operations

The fundamental operations performed on sets are:

Union
: A new set is obtained by combining the members of one set with the members of another set.

Intersection
: A new set is obtained by adding all the members of one set that also exist in a second set.

Difference
: A new set is obtained by adding all the members of one set except those that also exist in a second set.

The Set Class Implementation

The `Set` class implementation is built around an array for storing the data. We also create functions for each of the set operations outlined above. Here is the definition for the constructor function:

```
function Set() {
   this.dataStore = [];
   this.add = add;
   this.remove = remove;
   this.size = size;
   this.union = union;
   this.intersect = intersect;
   this.subset = subset;
   this.difference = difference;
   this.show = show;
}
```

Let's look at the `add()` function first:

```
function add(data) {
   if (this.dataStore.indexOf(data) < 0) {
      this.dataStore.push(data);
      return true;
   }
   else {
      return false;
   }
}
```

Because a set can only contain unique members, before the `add()` function can store data in the array, it must check to make sure the data isn't already in the array. We use the `indexOf()` function to check the array for the requested data. This function returns the position of an element in an array, or the value `-1` if the array doesn't contain the element. If the data isn't stored in the array, the function pushes the data onto the array and returns `true`. Otherwise, the function returns `false`. We need to write `add()` as a Boolean function so we have to way to know for sure whether or not the data was added to the set.

The `remove()` function works similarly to the `add()` function. We first check to see if the requested data is in the array. If it is, we call the `splice()` function to remove the data and return `true`. Otherwise, we return `false`, indicating the requested data isn't in the set. Here is the definition of `remove()`:

```
function remove(data) {
   var pos = this.dataStore.indexOf(data);
   if (pos > -1) {
      this.dataStore.splice(pos,1);
      return true;
   }
   else {
      return false;
   }
}
```

Before we can test these functions, let's define the `show()` function so we can see the members of a set:

```
function show() {
   return this.dataStore;
}
```

Example 9-1 demonstrates how the `Set` class works up to now.

Example 9-1. Using the `Set` class

```
load("set.js");
var names = new Set();
names.add("David");
names.add("Jennifer");
names.add("Cynthia");
names.add("Mike");
names.add("Raymond");
if (names.add("Mike")) {
   print("Mike added")
}
else {
   print("Can't add Mike, must already be in set");
}
print(names.show());
var removed = "Mike";
if (names.remove(removed)) {
   print(removed + " removed.");
}
else {
   print(removed + " not removed.");
}
names.add("Clayton");
print(names.show());
removed = "Alisa";
if (names.remove("Mike")) {
   print(removed + " removed.");
}
else {
   print(removed + " not removed.");
}
```

The output from Example 9-1 is:

```
Can't add Mike, must already be in set
David,Jennifer,Cynthia,Mike,Raymond
Mike removed.
David,Jennifer,Cynthia,Raymond,Clayton
Alisa not removed.
```

More Set Operations

The more interesting functions to define are `union()`, `intersect()`, `subset()`, and `difference()`. The `union()` function combines two sets using the union set operation to form a new set. The function first builds a new set by adding all the members of the first set. Then the function checks each member of the second set to see whether the member is already a member of the first set. If it is, the member is skipped over, and if not, the member is added to the new set.

Before we define `union()`, however, we need to define a helper function, `contains()`, which looks to see if a specified member is part of a set. Here is the definition for `contains()`:

```
function contains(data) {
   if (this.dataStore.indexOf(data) > -1) {
      return true;
   }
   else {
      return false;
   }
}
```

Now we can define the `union()` function:

```
function union(set) {
   var tempSet = new Set();
   for (var i = 0; i < this.dataStore.length; ++i) {
      tempSet.add(this.dataStore[i]);
   }
   for (var i = 0; i < set.dataStore.length; ++i) {
      if (!tempSet.contains(set.dataStore[i])) {
         tempSet.dataStore.push(set.dataStore[i]);
      }
   }
   return tempSet;
}
```

Example 9-2 demonstrates the use of `union()`:

Example 9-2. Computing the union of two sets

```
load("set.js");
var cis = new Set();
cis.add("Mike");
cis.add("Clayton");
cis.add("Jennifer");
cis.add("Raymond");
var dmp = new Set();
dmp.add("Raymond");
dmp.add("Cynthia");
dmp.add("Jonathan");
var it = new Set();
it = cis.union(dmp);
print(it.show());
//displays Mike,Clayton,Jennifer,Raymond,Cynthia,Jonathan
```

Set intersection is performed using a function named `intersect()`. This function is easier to define. Each time a member of the first set is found to be a member of the second set it is added to a new set, which is the return value of the function. Here is the definition:

```
    function intersect(set) {
       var tempSet = new Set();
       for (var i = 0; i < this.dataStore.length; ++i) {
          if (set.contains(this.dataStore[i])) {
             tempSet.add(this.dataStore[i]);
          }
       }
       return tempSet;
    }
```

Computing the intersection of two sets is shown in Example 9-3.

Example 9-3. Computing the intersection of two sets

```
load("set.js");
var cis = new Set();
cis.add("Mike");
cis.add("Clayton");
cis.add("Jennifer");
cis.add("Raymond");
var dmp = new Set();
dmp.add("Raymond");
dmp.add("Cynthia");
dmp.add("Bryan");
```

```
var inter = cis.intersect(dmp);
print(inter.show()); // displays Raymond
```

The next operation to define is subset. The `subset()` function first has to check to make sure that the proposed subset's length is less than the larger set we are comparing with the subset. If the subset length is greater than the original set, then it cannot be a subset. Once it is determined that the subset size is smaller, the function then checks to see that each member of the subset is a member of the larger set. If any one member of the subset is not in the larger set, the function returns `false` and stops. If the function gets to the end of the larger set without returning `false`, the subset is indeed a subset and the function returns `true`. The definition is below:

```
function subset(set) {
   if (this.size() > set.size()) {
      return false;
   }
   else {
      for each (var member in this.dataStore) {
         if (!set.contains(member)) {
            return false;
         }
      }
   }
   return true;
}
```

The `subset()` function uses the `size()` function before checking to see if each element of the sets match. Here is the code for the `size()` function:

```
function size() {
   return this.dataStore.length;
}
```

You'll notice that the `subset()` function uses a `for each` loop instead of a `for` loop, as we've used in the other definitions. Either loop will work here, but we just used the `for each` loop to show that its use is fine here.

Example 9-4 computes the subset of two sets.

Example 9-4. Computing the subset of two sets

```
load("set.js");
var it = new Set();
it.add("Cynthia");
it.add("Clayton");
it.add("Jennifer");
it.add("Danny");
it.add("Jonathan");
it.add("Terrill");
it.add("Raymond");
it.add("Mike");
```

```
var dmp = new Set();
dmp.add("Cynthia");
dmp.add("Raymond");
dmp.add("Jonathan");
if (dmp.subset(it)) {
   print("DMP is a subset of IT.");
}
else {
   print("DMP is not a subset of IT.");
}
```

Example 9-4 displays the following output:

```
DMP is a subset of IT.
```

If we add one new member to the `dmp` set:

```
dmp.add("Shirley");
```

then the program displays:

```
DMP is not a subset of IT.
```

The last operational function is `difference()`. This function returns a set that contains those members of the first set that are not in the second set. The definition for `difference()` is shown below:

```
function difference(set) {
   var tempSet = new Set();
   for (var i = 0; i < this.dataStore.length; ++i) {
      if (!set.contains(this.dataStore[i])) {
         tempSet.add(this.dataStore[i]);
      }
   }
   return tempSet;
}
```

Example 9-5 computes the difference of two sets.

Example 9-5. Computing the difference of two sets

```
load("set.js");
var cis = new Set();
var it = new Set();
cis.add("Clayton");
cis.add("Jennifer");
cis.add("Danny");
it.add("Bryan");
it.add("Clayton");
it.add("Jennifer");
var diff = new Set();
diff = cis.difference(it);
print("[" + cis.show() + "] difference [" + it.show()
      + "] -> [" + diff.show() + "]");
```

Example 9-5 displays:

```
[Clayton,Jennifer,Danny] difference [Bryan,Clayton,Jennifer] ->
[Danny]
```

Exercises

1. Modify the `Set` class so that the class stores its elements in sorted order. Write a program to test your implementation.
2. Modify the `Set` class so that it uses a linked list to store its elements rather than an array. Write a program to test your implementation.
3. Add the function `higher(element)` to the `Set` class. This function returns the least element in the set strictly greater than the given element. Test your function in a program.
4. Add the function `lower(element)` to the `Set` class. This function returns the greatest element in the set strictly less than the given element. Test your function in a program.