# 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. 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 written 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:

1. A set containing no members is called the *empty set*. The universe is the set of all possible members.
2. Two sets are considered equal if they contain exactly the same members.
3. 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:

1. *Union*: A new set is obtained by combining the members of one set with the members of another set.
2. *Intersection*: A new set is obtained by adding all the members of one set that also exist in a second set.
3. *Difference*: A new set is obtained by adding all the members of one set except those that also exist in a second set.

### Set Properties

The following properties are defined for sets:

1. The intersection of a set with the empty set is the empty set. The union of a set with the empty set is the original set.
2. The intersection of a set with itself is the original set. The union of a set with itself is theo original set.
3. Intersection and union are commutative. In other words, set1 intersect set2 is equal to set2 intersect set1, and the same is true for the union of two sets.
4. Intersection and union are associative. set1 intersect (set2 intersect set3) is equal to (set1 intersect set2) intersect set3. The same is true for the union of two sets.
5. The intersection of a set with the union of two other sets is distributive. In other words, set1 intersect (set2 union set3) is equal to (set1 intersect set2) union (set1 intersect set3).
6. The intersection of a set with the union of itself and another set yields the original set. This is also true for the union of a set with the intersection of itself and another set. This is called the *absorption law*.
7. The following equalities exist when the difference of the union or intersection of two sets is taken from another set:

* set1 difference (set2 union set3) equals (set1 difference set2) intersect (set1 difference set3)
* set1 difference (set2 intersect set3) equals (set1 difference set2) union (set1 difference set3)

The equalities are known as *DeMorgan's Laws*.

## The Set Class Implementation

Our Set class implementation is built around an array for storing the data. We also create methods 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;

}

A show() method is included so we can display the data stored in a set.

Let's look at the add() method 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() method can store data in the array it must check to make sure the data isn't already in the array. We use the indexOf() method to check the array for the requested data. If the data isn't stored in the array, the method pushes the data onto the array and returns true. Otherwise, the method returns false. We need to write add() as a Boolean method so we have to way to know for sure whether or not the data was added to the set.

The remove() method works similarly to the add() method. We first check to see if the requested data is in the array. If it is, we call the splice() method 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 methods, let's define the show() method so we can see the members of a set:

function show() {

return this.dataStore;

}

Now let's put these methods through their paces before we continue defining the class:

var names = new Set();

names.add("David");

names.add("Jennifer");

names.add("Cynthia");

names.add("Mike");

names.add("Raymond");

names.add("Mike");

print(names.show());

var removed = "Mike";

if (names.remove("Mike")) {

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 this program is:

David,Jennifer,Cynthia,Mike,Raymond

Mike removed.

David,Jennifer,Cynthia,Raymond,Clayton

Alisa not removed.

The more interesting methods to define are union(), intersect() (intersection), subset(), and difference(). The union() method combines two sets using the union set operation to form a new set. The method first builds a new set by adding all the members of the first set. Then the method 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 method, 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() method:

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;

}

Here is a program to test the union() method:

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 it = new Set();

it = cis.union(dmp);

print(it.show()); // displays Mike,Clayton,Jennifer,Raymond,Cynthia,Bryan

We perform set intersection using a method named intersect(). This method 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 method. 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;

}

The following program demonstrates how intersect() works, using the same sets from the previous example:

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 on the list is subset. The subset() method 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 set, then it cannot be a subset. Once it is determined that the subset size is smaller, the method 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 method returns false and stops. If the method gets to the end of the larger set without returning false, the subset is indeed a subset and the method 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;

}

You'll notice that the subset() method 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.

The program below makes use of the subset() method:

var it = new Set();

it.add("Cynthia");

it.add("Clayton");

it.add("Jennifer");

it.add("Danny");

it.add("Bryan");

it.add("Terrill");

it.add("Raymond");

it.add("Mike");

var dmp = new Set();

dmp.add("Cynthia");

dmp.add("Raymond");

dmp.add("Bryan");

if (dmp.subset(it)) {

print("DMP is a subset of IT.");

}

else {

print("DMP is not a subset of IT.");

}

The program displays:

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 method is difference(). This method 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;

}

Here is a program that demonstrates how difference() works:

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() + "]");

The output from this program is:

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

Formatting the output by putting brackets around the set members suggests that this is probably how the show() method should work, so let's redefine it:

function show() {

return "[" + this.dataStore + "]";

}