08. DATASTRUCTS 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.

8.1. 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.

8.2. 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.

8.3. 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.

8.4. 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:

Example: *Set* class constructor:

```
class Set {
   constructor(){
      this.dataStore = [];
   }
}
```

8.4.1 Adding a member to a Set.

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

Example: Set *add*() method:

```
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.

8.4.2. Removing a member from a 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*():

Example: Set *remove*() method:

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

8.4.3. Showing all members in a Set.

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

Example: Set *show*() method:

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

8.4.5. Testing Set Class Implementation.

Let's demonstrate how the Set class works up to now.

Example: Using the Set class.

```
// Testing program

var names = new Set();

// Adding members to the Set
names.add("David");
names.add("Jennifer");
names.add("Cynthia");
names.add("Mike");
names.add("Raymond");

// Adding Mike to the Set
if (names.add("Mike")) {
   console.log("Mike added");
}
else {
 console.log("Can't add Mike, must already be in set");
}
console.log(names.show());

// Removing Mike from the Set
var removed = "Mike";
if (names.remove(removed)) {
   console.log(removed + " removed.");
}
else {
 console.log(removed + " not removed.");
}

// Adding Clayton to the Set
names.add("Clayton");
print(names.show());

// Removing Alisa from the Set
removed = "Alisa";
if (names.remove("Mike")) {
   console.log(removed + " removed.");
}
else {
   console.log(removed + " not removed."); }
```

Output: Here is the output from the previous program.

```
Can't add Mike, must already be in set

David, Jennifer, Cynthia, Mike, Raymond

Mike removed.

David, Jennifer, Cynthia, Raymond, Clayton

Alisa not removed.
```

8.4.6. 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**():

Example:

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

Now we can define the union() function:

Example: Set *union*() method:

```
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;
     }
```

Let's test the union() function:

Example: Computing the union of two sets:

```
 // Adding memberts to 1st Set
var cis = new Set();
cis.add("Mike");
cis.add("Clayton");
cis.add("Jennifer");
cis.add("Raymond");

// Adding members to 2nd Set
var dmp = new Set();
dmp.add("Raymond");
dmp.add("Cynthia");
dmp.add("Jonathan");

// Union of the Sets
var it = new Set();
it = cis.union(dmp);

console.log(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:

Example:

```
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;
}
```

Let's test the *interset*() function:

Example: Computing the intersection of two sets:

```
 // Adding memberts to 1st Set
var cis = new Set();
cis.add("Mike");
cis.add("Clayton");
cis.add("Jennifer");
cis.add("Raymond");

// Adding members to 2nd Set
var dmp = new Set();
dmp.add("Raymond");
dmp.add("Cynthia");
dmp.add("Jonathan");

// Intersection of the Sets
var inter = new Set();
it = cis.intersect(dmp);

console.log(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:

Example: Set *subset*() method:

```
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:

Example: Set *size*() method:

```
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: Computing the subset of two sets.

```
// Adding memberts to 1st Set

 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");

// Adding memberts to 2nd Set

var dmp = new Set();

dmp.add("Cynthia");
dmp.add("Raymond");
dmp.add("Jonathan");


// Testing if DMP is a subset of IT

if (dmp.subset(it)) {
   console.log("DMP is a subset of IT.");
 }
 else {
   console.log("DMP is not a subset of IT.");
 }
```

Output: It displays the following output:

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

If we add one new member to the dmp set:

Example:

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

Output: 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:

Example: Set *difference*() method:

```
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: Computing the difference of two sets.

```
// Adding members to 1st Set

var cis = new Set();

cis.add("Clayton");
cis.add("Jennifer");
cis.add("Danny");

// Adding memberts to 2nd Set

var it = new Set();

it.add("Bryan");
it.add("Clayton");
it.add("Jennifer");


// Computing the difference of the sets

var diff = new Set();
diff = cis.difference(it);
console.log("[" + cis.show() + "] difference [" +
            it.show() + "] -> [" + diff.show() + "]");
```

Output: then the program displays:

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