Sets

Learn how to use sets, a data structure just introduced to JavaScript.

Introduction

Sets are a data structure commonly used in computer science.

A set is a collection of values. Think of it like a bucket. A set provides easy ways to insert and remove items. We can easily check if an item is present in our set and we can get all items back in a way that makes them easy to use.

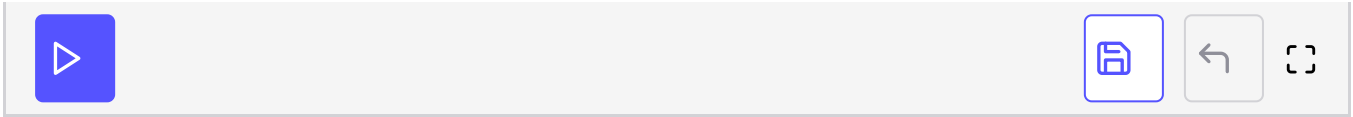
Creating a Set

We invoke a set using:

```
new Set();
```

Inside the constructor call, we can pass in an iterable object, usually an array. Every item in the array will be inserted into the set.

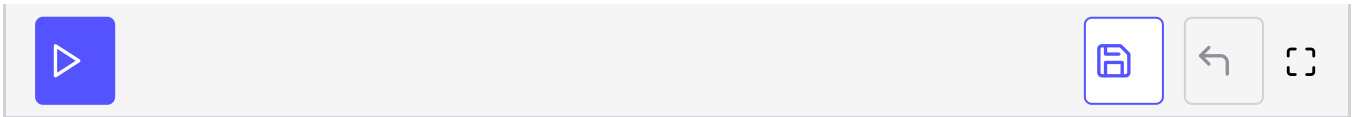
```
const set = new Set(['abc', 'def', 'ghi']);
console.log(set); // -> Set { 'abc', 'def', 'ghi' }
```



`new Set('string');`

One important caveat is that a string is also considered an iterable. If we call `new Set()` with a string, each individual character will be inserted into the set.

```
const set = new Set('abcde');
console.log(set); // -> Set { 'a', 'b', 'c', 'd', 'e' }
```

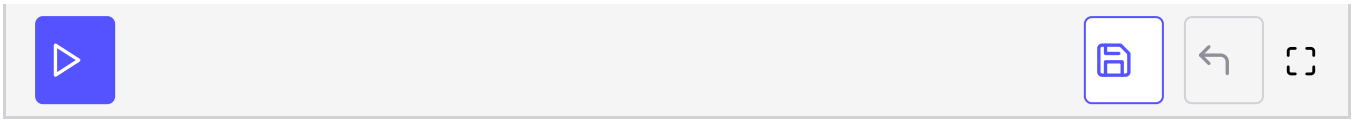


It's a common mistake to pass multiple arguments into the set constructor. This will not work. The set constructor takes zero or one arguments. All others will be ignored. The only way to pass in multiple items is to do so through an iterable parameter.

Basic Methods & Properties

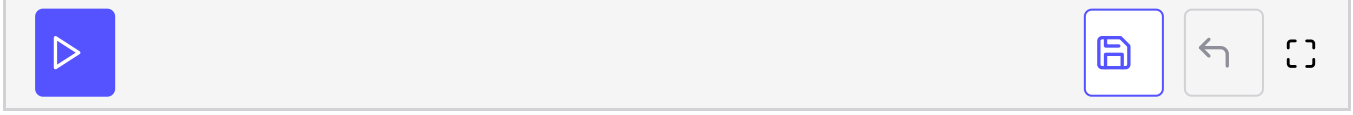
- `Set.add` inserts items
- `Set.delete` removes items
- `Set.has` checks if an item is present in the set
- `Set.size` is a property equal to the number of items in the set, similar to `Array.length`

```
const set = new Set();
set.add('abc');
set.add(17);

console.log(set); // -> Set { 'abc', 17 }
console.log(set.size); // -> 2

set.delete('abc');
console.log(set); // -> Set { 17 }

console.log(set.has(17)); // -> true
```



- `Set.clear` removes all items

```
const set = new Set();
set.add('abc');
set.add(17);
set.add(393);
set.add(false);

set.clear();
console.log(set); // -> Set {}
```

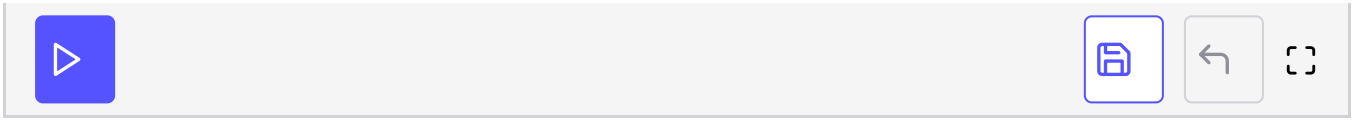


Insertion

Chaining Adds

The `add` function returns the set itself, so multiple `add` calls can be chained.

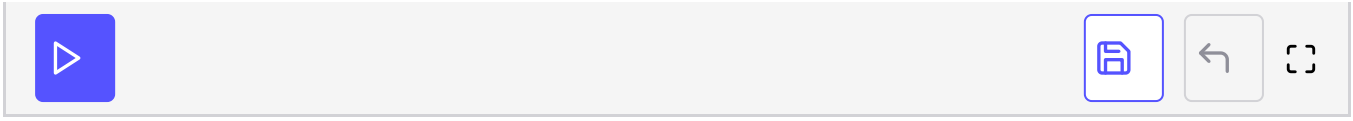
```
const set = new Set();
set.add(1).add(2).add(3);
console.log(set); // -> Set { 1, 2, 3 }
```



Duplicate Insertions

Duplicate insertions are completely ignored. This is one of the hallmarks of a set - it maintains one and only one copy of each item.

```
const set = new Set();

set.add('abc')
  .add(17)
  .add('abc')
  .add('abc');

console.log(set); // -> Set { 'abc', 17 }
```

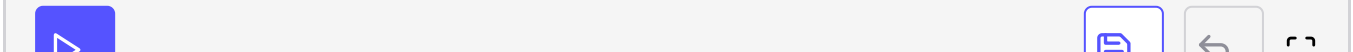


Inserting Objects

We can insert whatever we like into a set, including objects and arrays. We can even insert a set into a set.

Keep in mind that the set will maintain a *reference* to the object inserted. Therefore, when we check whether an object is present in a set or not, we are checking for the existence of one specific object. If we test using a clone of an object, `Set.has` will return `false`.

```
const set = new Set();
const obj = {};

set.add(obj);
console.log(set.has(obj)); // -> true
console.log(set.has({})); // -> false
```



Iterating

for-of loops

Sets can be iterated through using a for-of loop.

```
for(let item of set) {
}
```

`item` above is the variable that stores the items of the set in each iteration of the loop. Each item will be processed in the order in which it was inserted.

```
const set = new Set(['abc', 'def', 'ghi']);

for(let item of set) {
    console.log(item);
}
// -> abc
//    def
//    ghi
```

`Set.forEach`

Sets have their own `forEach` method that functions similar to the array counterpart.

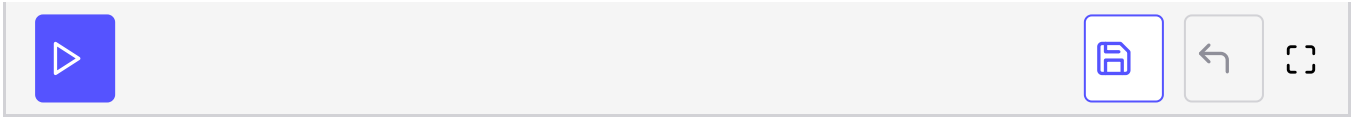
```
const set = new Set(['abc', 'def', 'ghi']);
set.forEach(item => console.log('Next item:', item));
// -> Next item: abc
// -> Next item: def
// -> Next item: ghi
```

Arguments Provided

You may recall that the array methods pass two other parameters into their callbacks: the index and the array itself. Sets don't use indexes, so instead of the index as the second argument, they just provide the value again. The last

argument is the whole set.

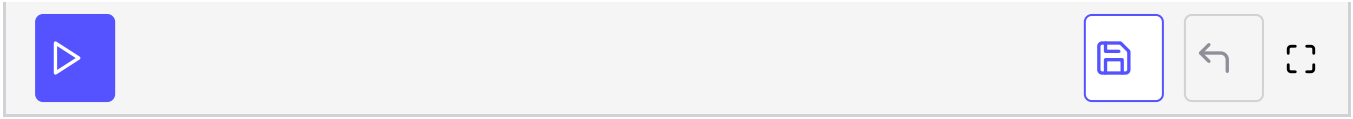
```
const set = new Set(['abc', 'def', 'ghi']);

set.forEach((item, itemAgain, originalSet) => {
    console.log(item, itemAgain, originalSet);
});

// -> abc abc Set { 'abc', 'def', 'ghi' }
// -> def def Set { 'abc', 'def', 'ghi' }
// -> ghi ghi Set { 'abc', 'def', 'ghi' }
```



Using `_`

There's almost no reason we'd need the same argument twice in an array. To show that we want to ignore the second argument, we can use an underscore `_` as its variable.

```
const set = new Set(['abc', 'def', 'ghi']);
set.forEach((item, _, originalSet) => {
    // do something
});
```



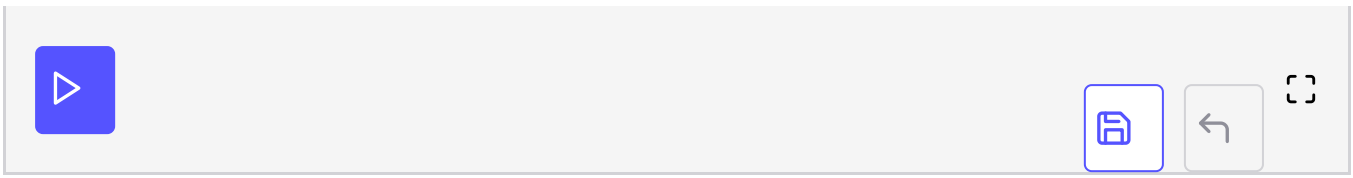
Set Operations

Sets have `keys`, `values`, and `entries` methods, just like objects. These methods have some quirks.

Keys and Values

`Set.keys` and `Set.values` are the exact same function. They return the items present in the set in the same order in which they were inserted.

```
const set = new Set();

console.log(set.keys === set.values); // -> true

set.add('abc')
  .add([17])
  .add(false);

console.log(set.values()); // -> SetIterator { 'abc', [ 17 ], false }
```

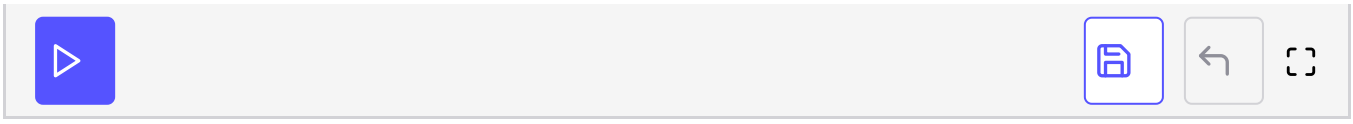


You'll notice that the item returned by `Set.keys` is a `SetIterator`. What this means is that internally, the set maintains the order of our data.

Entries

`Set.entries` is similar to `keys` and `values`. It attempts to mimic `Object.entries`.

```
const set = new Set(['abc', 'def', 'ghi']);
console.log(set.entries());
// -> SetIterator { [ 'abc', 'abc' ], [ 'def', 'def' ], [ 'ghi', 'ghi' ] }
```



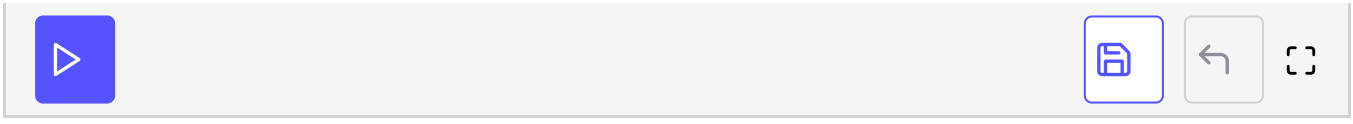
Instead of providing the items in the set, it maps each item into an array that contains each item twice and returns that. It's not very useful.

Of these three methods, all you'll need is one of `Set.keys` and `Set.values`. Use whichever one you like.

Sets into Arrays

A set can be easily transformed into an array for easier manipulation of values using the spread operator.

```
const set = new Set(['abc', 'def', 'ghi']);
const arr = [...set];
console.log(arr); // -> [ 'abc', 'def', 'ghi' ]
```



This is done frequently when working with the data in a set.

Extended Functionality

In traditional implementations, sets often have other useful methods attached

to them. These include functions that operate on multiple sets. They might perform unions, check if a set is a subset of another, or find the difference between sets.

Unfortunately, JavaScript sets don't have these methods and so their use is rather limited. To check out more methods, look at MDN's page. They provide examples of how these functions might be implemented.

That's it for sets.