# Hash Tables

Save    Edit on GitHub

A hash table is an implementation of an associative array, a list of key-value pairs that allow you to retrieve a value via a key. Internally a hash table utilizes a hash function to transform a key value into an index that points to where the value is stored in memory. Hash tables have fast search, insertion and delete operations.

There are two main ways to implement a hash table/associative array in JavaScript.

## Using the Object Data Type

The simplest implementation is using the `Object` data type. This is because all non-scalar objects in JavaScript behave as associative arrays, a mapping from property keys to values. So an `Object` itself can behave as a basic hash table.

```
var simplehash = new Object();
// or
// var simplehash = {};

simplehash['key1'] = 'value1';
simplehash['key2'] = 'value2';
simplehash['key3'] = 'value3';

for (var key in simplehash) {
  // use hasOwnProperty() to filter out properties from Object.prototype
  if (simplehash.hasOwnProperty(key)) {
    console.log('key is: ' + key + ', value is: ' + simplehash[key]);
  }
}
```

The output would look like:

```
key is: key1, value is: value1
key is: key2, value is: value2
key is: key3, value is: value3
```

There are some downsides to this approach:

- The `Object` comes with its own properties which could collide with potential key names.
- There no easy way to get the size of a Hash Table stored in an `Object`, so it must be tracked manually.
- Since they are also property names, the keys used are limited to `String` or `Symbol` types.
- `Object` isn't optimized for frequent additions and removals of key-value pairs

## Using a Map Object

The `Map` object was created to implement this type of associative array without some of the downsides of using a basic `Object`:

- There are no pre-existing keys that could result in a collision
- A `Map` object has a `size` property to track its contents.
- A `Map` object can have keys that are any data type.
- A `Map` has been optimized for repeated addition and insertion of key-value pairs.

A `Map` object also comes with the following methods:

- `.clear()` Removes all key-value pairs from the `Map` object.
- `.delete(key)` Deletes the key-value pair and returns `true` if the key exists. Returns `false` otherwise.
- `.get(key)` Returns the value associated with key, or `undefined` if key doesn't exist.
- `.has(key)` Returns `true` if key exists, `false` otherwise.
- `.set(key,value)` Sets the value for the key in the `Map` object and returns the `Map` object.

**Note:** Key-value pairs must be set with the `set` method in order for the `Map` object to behave as expected. Using the syntax for `Object` above will appear to work, but will not associate the key-value pair to its internal collection.

```
var maphash = new Map();

maphash.set('key1', 'value1');
maphash.set('key2', 'value2');
maphash.set('key3', 'value3');

console.log(maphash.get('key3'));
// Output: value3

maphash.set('key1', 'new value');

console.log(maphash.get('key1'));
// Output: new value

console.log(maphash.size);
// Output: 3

maphash.delete('key2');

console.log(maphash.size);
// Output: 2

for (const [key, value] of maphash) {
  console.log(key + ' = ' + value);
}
// Output: key1 = new value
//         key3 = value3
```