WeakMaps

In order to store private data versions < ES6, we had various ways of doing this. One such method was using naming conventions:

```
class Person {
    constructor(age) {
        this._age = age;
    }

    _incrementAge() {
        this._age += 1;
    }
}
```

But naming conventions can cause confusion in a codebase and are not always going to be upheld. Instead, we can use WeakMaps to store our values:

```
let _age = new WeakMap();
class Person {
    constructor(age) {
        _age.set(this, age);
    }

    incrementAge() {
        let age = _age.get(this) + 1;
        _age.set(this, age);
        if (age > 50) {
            console.log('Midlife crisis');
        }
    }
}
```

The cool thing about using WeakMaps to store our private data is that their keys do not give away the property names, which can be seen by using `Reflect.ownKeys()`:

```
> const person = new Person(50);
> person.incrementAge(); // 'Midlife crisis'
> Reflect.ownKeys(person); // []
```

A more practical example of using WeakMaps is to store data which is associated to a DOM element without having to pollute the DOM itself:

```
let map = new WeakMap();
let el  = document.getElementById('someElement');

// Store a weak reference to the element with a key
map.set(el, 'reference');

// Access the value of the element
let value = map.get(el); // 'reference'

// Remove the reference
el.parentNode.removeChild(el);
el = null;

value = map.get(el); // undefined
```

As shown above, once the object is destroyed by the garbage collector, the WeakMap will automatically remove the key-value pair which was identified by that object.

> **Note**: To further illustrate the usefulness of this example, consider how jQuery stores a cache of objects corresponding to DOM elements which have references. Using WeakMaps, jQuery can automatically free up any memory that was associated with a particular DOM element once it has been removed from the document. In general, WeakMaps are very useful for any library that wraps DOM elements.