Singleton Pattern

Share a single global instance throughout our application

Singletons are classes which can be instantiated once, and can be accessed globally. This single instance can be shared throughout our application, which makes Singletons great for managing global state in an application.

First, let's take a look at what a singleton can look like using an ES2015 class. For this example, we're going to build a `Counter` class that has:

```
let counter = 0;

class Counter {
  getInstance() {
    return this;
  }

  getCount() {
    return counter;
  }

  increment() {
    return ++counter;
  }

  decrement() {
    return --counter;
  }
}
```

a `getInstance` method that returns the value of the instance

a `getCount` method that returns the current value of the `counter` variable

an `increment` method that increments the value of `counter` by one

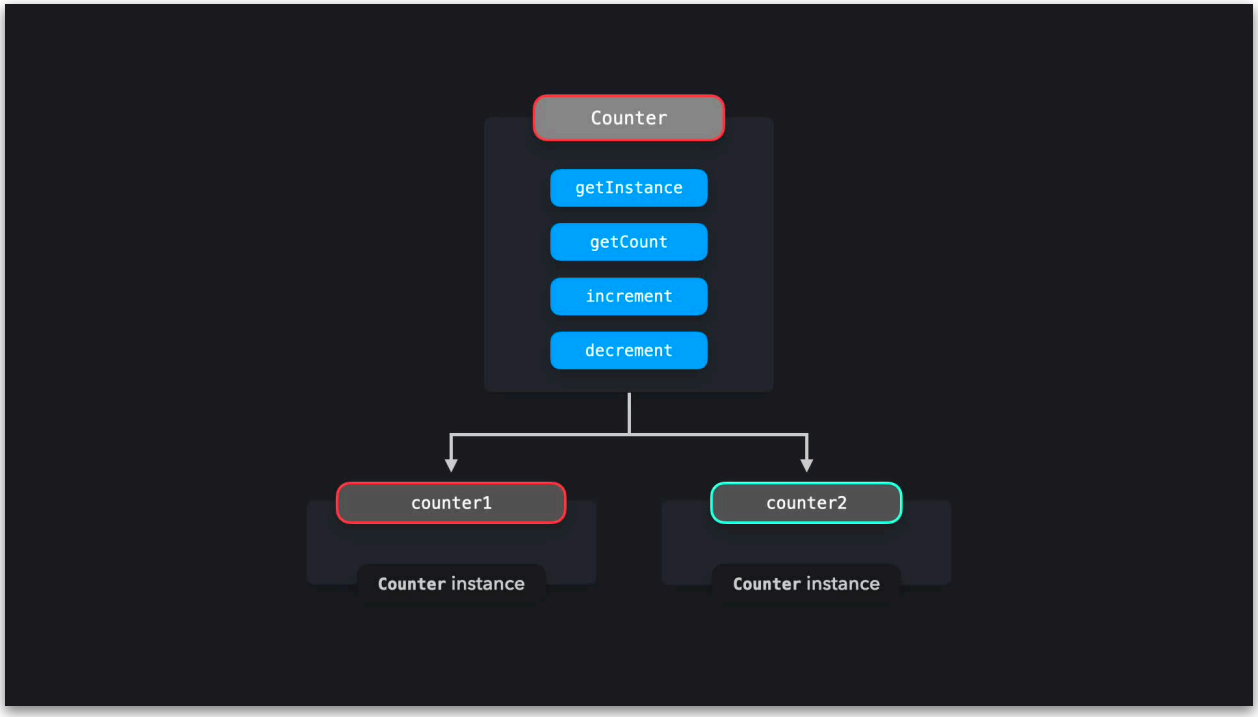
a `decrement` method that decrements the value of `counter` by one

However, this class doesn't meet the criteria for a Singleton! A Singleton should only be able to get instantiated once. Currently, we can create multiple instances of the `Counter` class.

```
let counter = 0;

class Counter {
  getInstance() {
    return this;
  }

  getCount() {
    return counter;
  }

  increment() {
    return ++counter;
  }

  decrement() {
    return --counter;
  }
}

const counter1 = new Counter();
const counter2 = new Counter();

console.log(counter1.getInstance() === counter2.getInstance()); // false
```

By calling the new method twice, we just set `counter1` and `counter2` equal to different instances. The values returned by the getInstance method on `counter1` and `counter2` effectively returned references to different instances: they aren't strictly equal!

Let's make sure that only one instance of the `Counter` class can be created.



One way to make sure that only one instance can be created, is by creating a variable called instance. In the constructor of `Counter`, we can set instance equal to a reference to the instance when a new instance is created. We can prevent new instantiations by checking if the instance variable already had a value. If that's the case, an instance already exists. This shouldn't happen: an error should get thrown.

```
let instance;
let counter = 0;

class Counter {
  constructor() {
    if (instance) {
      throw new Error("You can only create one instance!");
    }
    instance = this;
  }

  getInstance() {
    return this;
  }

  getCount() {
    return counter;
  }

  increment() {
    return ++counter;
  }

  decrement() {
    return --counter;
  }
}

const counter1 = new Counter();
const counter2 = new Counter();
// Error: You can only create one instance!
```

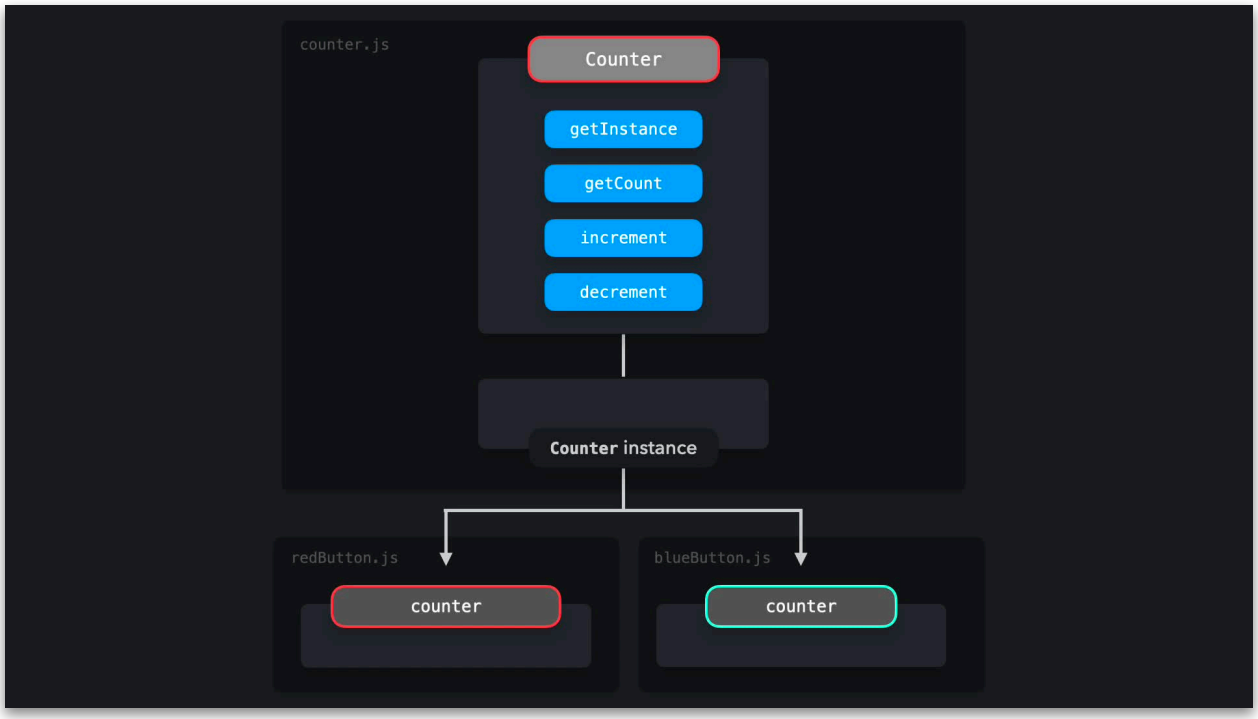
Perfect! We aren't able to create multiple instances anymore.

Let's export the `Counter` instance from the `counter.js` file. But before doing so, we should freeze the instance as well.
The `Object.freeze` method makes sure that consuming code cannot modify the Singleton. Properties on the frozen instance cannot be added or modified, which reduces the risk of accidentally overwriting the values on the Singleton.

```
let instance;
let counter = 0;

class Counter {
  constructor() {
    if (instance) {
      throw new Error("You can only create one instance!");
    }
    instance = this;
  }

  getInstance() {
    return this;
  }

  getCount() {
    return counter;
  }

  increment() {
    return ++counter;
  }

  decrement() {
    return --counter;
  }
}

const singletonCounter = Object.freeze(new Counter());
export default singletonCounter;
```

Let's take a look at an application that implements the Counter example. We have the following files:

- `counter.js`: contains the `Counter` class, and exports a `Counter` instance as its default export
- `index.js`: loads the `redButton.js` and `blueButton.js` modules
- `redButton.js`: imports `Counter`, and adds `Counter`'s `increment` method as an event listener to the red button, and logs the current value of counter by invoking the `getCount` method
- `blueButton.js`: imports Counter, and adds Counter's increment method as an event listener to the blue button, and logs the current value of counter by invoking the getCount method



Both `blueButton.js` and `redButton.js` import the same instance from `counter.js`. This instance is imported as `Counter` in both files.

When we invoke the increment method in either `redButton.js` or `blueButton.js`, the value of the counter property on the `Counter` instance updates in both files. It doesn't matter whether we click on the red or blue button: the same value is shared among all instances. This is why the counter keeps incrementing by one, even though we're invoking the method in different files.

(Dis)advantages

Restricting the instantiation to just one instance could potentially save a lot of memory space. Instead of having to set up memory for a new instance each time, we only have to set up memory for that one instance, which is referenced throughout the application. However, Singletons are actually considered an anti-pattern, and can (or.. should) be avoided in JavaScript.

In many programming languages, such as Java or C++, it's not possible to directly create objects the way we can in JavaScript. In those object-oriented programming languages, we need to create a class, which creates an object. That created object has the value of the instance of the class, just like the value of instance in the JavaScript example.

However, the class implementation shown in the examples above is actually overkill. Since we can directly create objects in JavaScript, we can simply use

a regular object to achieve the exact same result. Let's cover some of the disadvantages of using Singletons!

Using a regular object

Let's use the same example as we saw previously. However this time, the counter is simply an object containing:

- a `count` property
- an `increment` method that increments the value of `count` by one
- a `decrement` method that decrements the value of `count` by one

```
let count = 0;

const counter = {
  increment() {
    return ++count;
  },
  decrement() {
    return --count;
  }
};

Object.freeze(counter);
export { counter };
```

Since objects are passed by reference, both `redButton.js` and `blueButton.js` are importing a reference to the same singleton `Counter` object. Modifying the value of count in either of these files will modify the value on the singletonCounter, which is visible in both files.

Testing

Testing code that relies on a Singleton can get tricky. Since we can't create new instances each time, all tests rely on the modification to the global instance of the previous test. The order of the tests matter in this case, and one small modification can lead to an entire test suite failing. After testing, we need to reset the entire instance in order to reset the modifications made by the tests.

```
import Counter from "../src/counterTest";

test("incrementing 1 time should be 1", () => {
  Counter.increment();
  expect(Counter.getCount()).toBe(1);
});

test("incrementing 3 extra times should be 4", () => {
  Counter.increment();
  Counter.increment();
  Counter.increment();
  expect(Counter.getCount()).toBe(4);
});

test("decrementing 1  times should be 3", () => {
  Counter.decrement();
  expect(Counter.getCount()).toBe(3);
});
```

Dependency hiding

When importing another module, superCounter.js in this case, it may not be obvious that that module is importing a Singleton. In other files, such as index.js in this case, we may be importing that module and invoke its methods. This way, we accidentally modify the values in the Singleton. This can lead to unexpected behavior, since multiple instances of the Singleton can be shared throughout the application, which would all get modified as well.

index.js

```
import Counter from "./counter";
import SuperCounter from "./superCounter";
const counter = new SuperCounter();

counter.increment();
counter.increment();
counter.increment();

console.log("Counter in index.js", Counter.getCount())
```

Global behavior

A Singleton instance should be able to get referenced throughout the entire app. Global variables essentially show the same behavior: since global variables are available on the global scope, we can access those variables throughout the application.

Having global variables is generally considered as a bad design decision. Global scope pollution can end up in accidentally overwriting the value of a global variable, which can lead to a lot of unexpected behavior.

In ES2015, creating global variables is fairly uncommon. The new let and const keyword prevent developers from accidentally polluting the global scope, by keeping variables declared with these two keywords block-scoped. The new module system in JavaScript makes creating globally accessible values easier without polluting the global scope, by being able to export values from a module, and import those values in other files.

However, the common usecase for a Singleton is to have some sort of global state throughout your application. Having multiple parts of your codebase rely on the same mutable object can lead to unexpected behavior.

Usually, certain parts of the codebase modify the values within the global state, whereas others consume that data. The order of execution here is important: we don't want to accidentally consume data first, when there is no data to consume (yet)! Understanding the data flow when using a global state can get very tricky as your application grows, and dozens of components rely on each other.

State management in React

In React, we often rely on a global state through state management tools such as Redux or React Context instead of using Singletons. Although their global state behavior might seem similar to that of a Singleton, these tools provide a read-only state rather than the mutable state of the Singleton. When using

Redux, only pure function reducers can update the state, after a component has sent an action through a dispatcher.

Although the downsides to having a global state don't magically disappear by using these tools, we can at least make sure that the global state is mutated the way we intend it, since components cannot update the state directly.