Mixins

introduction to inheritance and the importance of mixins in JS; and cloning objects in ES6

Heated debates of composition over inheritance made mixins appear to be the winner construct for composing objects. Therefore, libraries such as UnderscoreJs and LoDash created support for this construct with their methods `_.extend` or `_.mixin`.

In ES6, `Object.assign` does the same thing as `_.extend` or `_.mixin`.

Why are mixins important?

They are important because the alternative of establishing a class hierarchy using inheritance is inefficient and rigid.

Suppose you have a view object, which can be defined with or without the following extensions:

- validation
- tooltips
- abstractions for two-way data binding
- toolbar
- preloader animation

Assuming the order of the extensions does not matter, 32 different view types can be defined using the five enhancements above. To fight the combinatoric explosion, we take these extensions as mixins and extend our object prototypes with the extensions that we need.

For instance, a validating view with a preloader animation can be defined in the following way:

```
let View = { ... };
let ValidationMixin = { ... };
```

```
let PreloaderAnimationMixin = { ... };

let ValidatingMixinWithPreloader = Object.assign(
    {},
    View,
    ValidationMixin,
    PreloaderAnimationMixin
);
```

Why do we extend the empty object?

Because `Object.assign` works in a way that it extends its first argument with the remaining list of arguments. This implies that the first argument of `Object.assign` may get new keys, or its values will be overwritten by a value originating from a mixed in object.

Syntax for `Object.assign`

The syntax for calling `Object.assign` is as follows:

> `Object.assign( targetObject, ...sourceObjects )`

The return value of `Object.assign` is `targetObject`. The side-effect of calling `Object.assign` is that `targetObject` is mutated.

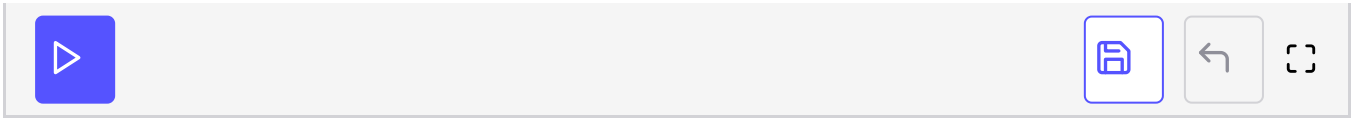
> `Object.assign` makes a *shallow copy* of the properties and operations of `...sourceObjects` into `targetObject`.

For more information on *shallow copies* or cloning, check my article on Cloning Objects in JavaScript, and check the first exercise of the lesson on the Spread operator and Rest parameters.

Consider the following code:

```
let horse = {
        horseName: 'QuickBucks',
        toString: function() {
            return this.horseName;
        }
};

let rider = {
        riderName: 'Frank',
        toString: function() {
```

```
            return this.riderName;
        }
};

let horseRiderStringUtility = {
    toString: function() {
        return this.riderName + ' on ' + this.horseName;
    }
}

let racer = Object.assign(
        {},
        horse,
        rider,
        horseRiderStringUtility
    );

console.log( racer.toString() );
```



Had we omitted the `{}` from the assembly of the `racer` object, seemingly, nothing would have changed, as `racer.toString()` would still have been `"Frank on QuickBucks"`. However, notice that `horse` would have been `===` equivalent to `racer`, meaning, that the side-effect of executing `Object.assign` would have been the mutation of the `horse` object.