The Spread Operator and Objects

In this lesson you will use the spread operator on an object.

WE'LL COVER THE FOLLOWING

Simple copy

You can spread an object similar to an array. Spreading an object is handy in a scenario where you need to create a clone for immutability purposes or if you are already using the function `Object.assign`.

The `assign` method is similar in functionality but uses a more verbose syntax. The syntax for spread operator is identical to the spread for an array which means that it uses the three dots in front of the object instead of the array.

To create a clone, you need to define a variable and equal open curly bracket, spread operator, the variable to copy, and close the curly bracket. It copies all members and has them *floating* side by side with a comma. Indeed, this is just a visualization to help you illustrate that the curly brackets take the result and form a new object.

In the following code, line 2 is performing the object clone. Notice the curly brackets `{` and `}` and the use of the spread operator `...`. The output is `false` because they are not the same object, they have a different reference in memory. Altering one, will not change the other.

```
const objToClone = { prop1: 1, prop2: "2" };
const clonedObject = { ...objToClone };

console.log(objToClone === clonedObject); // Not the same
```

Shallow copy

Spreading an object will return a *shallow copy* of all members of the object. It means that it does not create a copy of an object inside the object, only the primitive types.

```
const objToClone = { prop1: 1, prop2: "2", innerObject: { prop2: "3" } };
const clonedObject = { ...objToClone };

console.log(objToClone.innerObject === clonedObject.innerObject); // Same
```

Copying and adding members

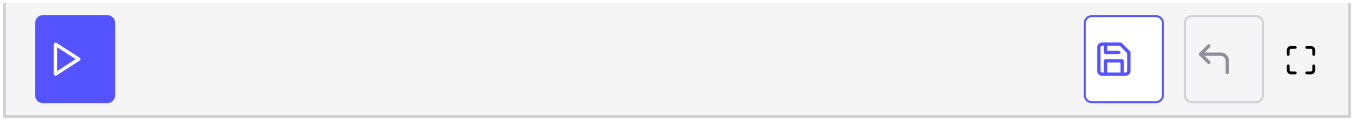
Spreading can go a step further. Imagine that you want to **add** value or multiple values to a new cloned object. It is possible by declaring a new variable, assigning a new object by opening the curly bracket and spreading the object with all the value you have.

Instead of closing the curly bracket right away, which would just create a *shallow clone*, insert a comma and type the name of the property you want to assign to the variable followed by a colon and the value. Close the curly bracket.

The spread operator will set all members and values, and proceed by setting the members defined after the spread. An example is worth a thousand words, glance at **line 3** to see how `prop3` and `prop4` are added to the new object.

```
const objToClone = { prop1: 1, prop2: "2" };
```

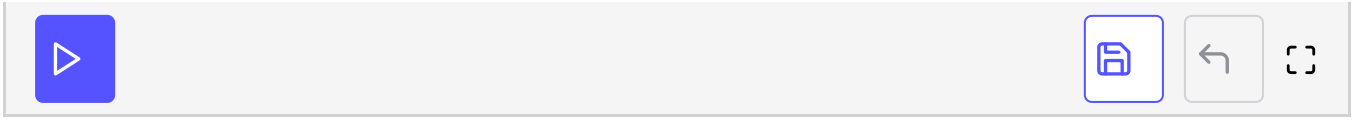
```
const clonedObject = { ...objToClone };
const clonedObjectWithMore = { ...objToClone, prop3: "3", prop4: false };
console.log(clonedObjectWithMore);
```



Copy with an override value

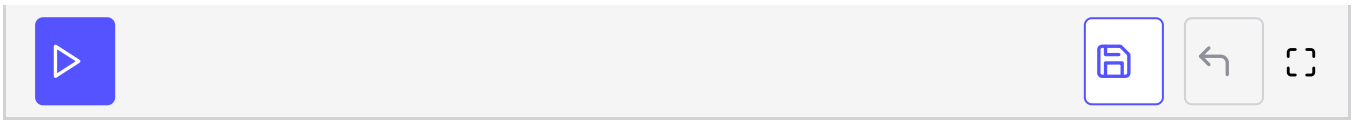
If the value existed in both the spread object and the one manually defined, the latter one would prevail. The cascade is a trick that is often used to have a set of default values and to let the user override some of the configurations. You could spread a default object followed by a comma and spreading an object with some properties. The result would be a full object with default values where the specified user values have precedence by overriding the default values.

```
const objToClone = { prop1: 1, prop2: "2" };
const clonedObjectWithMore2 = { ...objToClone, prop2: "Override" };
console.log(clonedObjectWithMore2);
```



Here is an example illustrating the default value.

```
const defaultValue = { id: -1, name: "unknown" };
const properties = { ...defaultValue, name: "John Doe" };
console.log(properties); // Will have the default -1
```



Spreading an object literal

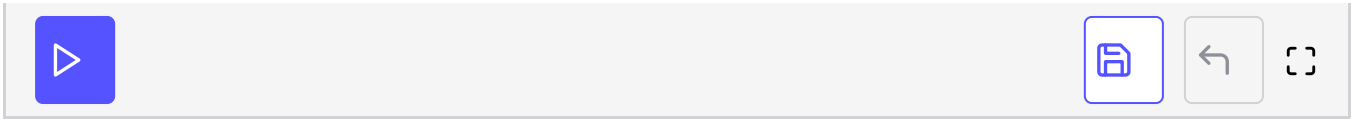
A final detail to understand is that spreading an object **only returns** the object's own enumerable properties. The property constraint is important to understand because a rich object (with functions) will lose them.

However, if you are manipulating a data object, everything is cloned. A framework like Redux promotes having an immutable store which is formed by many data objects. Cloning with the spread operator is a way to have immutability.

The following code demonstrates that a literal object with a function will be spread but an object from a class won't be.

Lines 1 to 7 defines an object that has a function. The cloning of the object at **line 8** copies the object. Then, the function can be used. One remark is that both functions remain the same, see the output of **line 11**. Nonetheless, **line 13** will output the property of each object because the `this` of the function points to the respective object.

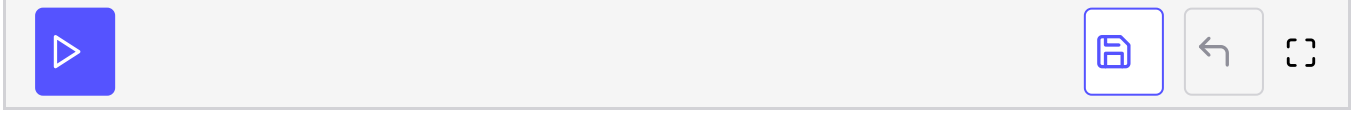
```
const objToCloneWithFunction = {
    prop1: 1,
    prop2: "2",
    funct1: function () {
        console.log("Prop1:", this.prop1);
    },
};
const clonedObjectWithoutFunc = { ...objToCloneWithFunction };
console.log(clonedObjectWithoutFunc); //{ prop1: 1, prop2: '2', funct1: [Function: funct1] }
clonedObjectWithoutFunc.funct1(); // Print 1
console.log(clonedObjectWithoutFunc.funct1 === objToCloneWithFunction.funct1);
objToCloneWithFunction.prop1 = 10000;
objToCloneWithFunction.funct1(); // Print 10000
```



Spreading an object from a class

However, if the object is coming from a `class` instead of an object literal, the behavior is different. In the following code, **line 9** does not have the `funct1`.

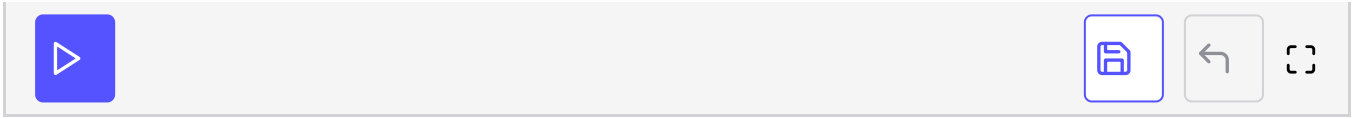
```
class ClassToClone {
    public funct1() {
        console.log("Props1:", this.prop1);
    }
    constructor(public prop1: number, public prop2: string) {}
}
const classToClone = new ClassToClone(1, "2");
const classCloned = { ...classToClone };
console.log(classCloned); //{ prop1: 1, prop2: '2' }
```



A small modification to the class by **adding a function into a variable** will

make this one available to the cloned object.

```
class ClassToClone2 {
  public funct2 = () => { console.log("Props1:", this.prop1); }
  constructor(public prop1: number, public prop2: string) { }
}
const classToClone2 = new ClassToClone2(1, "2");
const classCloned2 = { ...classToClone2 };
console.log(classCloned2); // { prop1: 1, prop2: '2', funct2: [Function] }
```



The reason is that the function of the class goes into the prototype, and as mentioned, only members of the object (not the prototype chain) get spread.