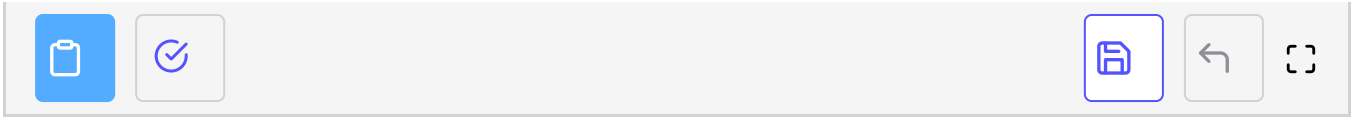
Exercise on Objects in ES6

Now, we'll create a "Basket" object using all the concepts we have learned in this section.

Exercise 1:

Suppose an array of `firstName`, `email`, `basketValue` triples are given. Create ONE JavaScript expression that puts a default value of `'-'` and `0` to the `firstName` or `basketValue` fields respectively, whenever the `firstName` or the `basketValue` keys are missing. Remember to name your new object, `newBaskets` or your code won't run.

```
// the baskets object is randomly generated
console.log("baskets: ");
console.log(baskets);

let newBaskets = {}
```



Explanation

The hard part of the solution is that we need one JavaScript expression. When it comes to transforming the value of an array, we usually use the map method. As we learned in the first lesson, the shortest way of writing a map function is through using the arrow syntax. Inside the map, we have to fill in the default values in item. If a key is already given in item, it takes precedence. We will use `Object.assign`. For each element of the basket array, we will create an object of default values, and we extend it with the element of the array, mixing in all the properties. If a key exists in both objects, the value in `item` is kept.

Exercise 2:

Create a prototype object with the following methods:

- `addToBasket( value )` adds `value` to the basket value

- `clearBasket()` sets the basket value to 0
- `getBasketValue()` returns the basket value
- `pay()` logs the message `{getBasketValue()} has been paid`, where `{getBasketValue()}` is the return value of the method with the same name. We can pay for the same basket as many times as we'd like. Name your object, `basketProto` your code won't compile otherwise.

```
let basketProto = {
  //write-your-code-here
}
```

Exercise 3:

Create an object `myBasket`, and set its prototype to the object created in Exercise 2 (it has already been prepended to the given code). Create an array field in `myBasket`, containing all the items that you purchase in the following format:

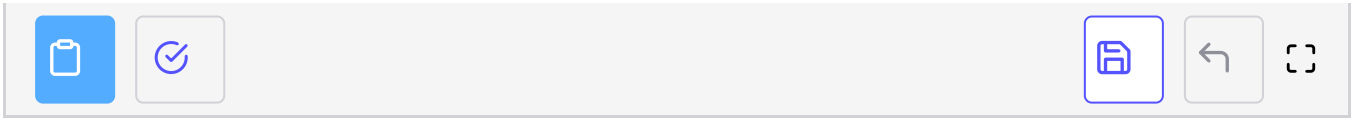
```
{ itemName: 'string', itemPrice: 9.99 }
```

Redefine the `addToBasket` method such that it accepts an `itemName` and an `itemPrice`. Call the `addToBasket` method in the prototype for the price administration, and store the `itemName`-`itemPrice` data locally in your array. Make sure you modify the `clearBasket` method accordingly.

```
myBasket.addToBasket( 'Cream', 5 );
myBasket.addToBasket( 'Cake', 8 );
myBasket.getBasketValue();            // 13
myBasket.items
// [{ itemName: 'Cream', itemPrice: 5},
//  { itemNAme: 'Cake', itemPrice: 8 }]
myBasket.clearBasket();
myBasket.getBasketValue();            // 0
myBasket.items                        // []
```

```
//basketProto object given
let basketProto = {
  value: 0
```

```
  addToBasket( itemValue ){
    this.value += itemValue;
  },
  clearBasket() {
    this.value = 0;
  },
  getBasketValue(){
    return this.value;
  },
  pay() {
    console.log( this.getBasketValue() + ' has been paid' );
  }

};

//write-your-code-here
```



Explanation

Notice,

- the shorthand of constructing `{ itemName, itemPrice }`,
- the short method syntax enabling the super calls,
- the easy way of setting prototypes.

ES6 is powerful and clean.

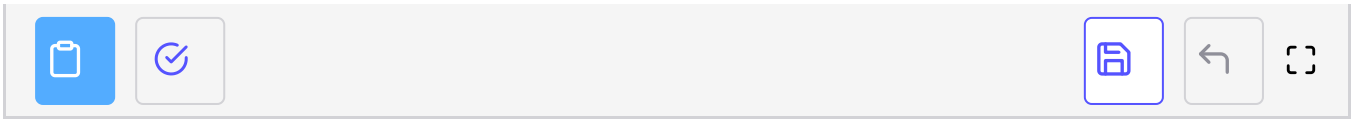
Exercise 4:

Extend your solution in Exercise 3 by adding a `removeFromBasket( index )` method. The parameter `index` should be the index of the element in the array that you would like to remove.

```
myBasket.addToBasket( 'Cream', 5 );
myBasket.addToBasket( 'Cake', 8 );
myBasket.addToBasket( 'Cookie', 2 );
myBasket.removeFromBasket( 1 );

myBasket.getBasketValue();            // 7
myBasket.items
// [{ itemName: 'Cream', itemPrice: 5},
//  { itemNAme: 'Cookie', itemPrice: 2 }]
```

```
//basketProto object given
```

```
let basketProto = {
  value: 0,
  addToBasket( itemValue ){
    this.value += itemValue;
  },
  clearBasket() {
    this.value = 0;
  },
  getBasketValue(){
    return this.value;
  },
  pay() {
    console.log( this.getBasketValue() + ' has been paid' );
  }

};

let myBasket = {
  items: [],
  addToBasket( itemName, itemPrice )
  {
    this.items.push( { itemName, itemPrice } );
    super.addToBasket( itemPrice );
   },
   clearBasket() {
this.items = [];
super.clearBasket(); }
};
Object.setPrototypeOf( myBasket, basketProto );
```



The hard part of this task is synchronizing the value stored in the prototype. By the time you reach Exercise 4, you might have noticed that this redundancy is not optimal from the perspective of modeling the baskets.

The index needs to be valid. For simplicity, I have omitted to check for floating point indices. Notice that it is not possible to use arrow functions here, because of the properties of fat arrows you learned in Lesson 1. Using arrow functions preserves the external context, which is not `myBasket`. Therefore, accessing this would not give us the desired results. The call `A.splice( index, 1 )` method removes `A[index]` from `A` mutating the original array, and returns an array of the removed elements. As we only removed one element, we find our element at index 0.

We already got used to the handy super call in Exercise 3. I added this method as an extension of `myBasket` on purpose to show you that the super call cannot be used with this function syntax. In fact, this form is discouraged. Use the concise method syntax, and define all methods of an object at once. In the

unlikely case you still needed this format, you would have to get the prototype

of the current object, and call its method by setting the context. With the concise method syntax, our solution would look like this:

```
let myBasket = { items: [],
addToBasket( itemName, itemPrice ){ this.items.push( { itemName, itemPrice } ); super.addToBa
    },
    clearBasket() {
  this.items = [];
  super.clearBasket(); },
  removeFromBasket( index ) {
  if ( typeof index !== 'number' ||
  index < 0 ||
  index >= this.items.length ) return;
  let removedElement = this.items.splice( index, 1 )[0]; super.addToBasket( -removedElement.i
  } };
```