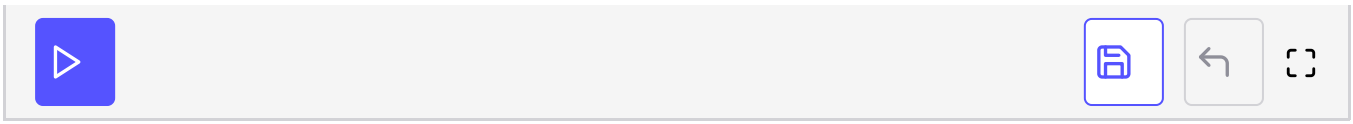
Object.getOwnPropertyDescriptors

the behavior of value, writable, get, set, configurable and enumerable

`Object.getOwnPropertyDescriptors` returns all property descriptors of its first argument:

```
let player = {
    cards: [ 'Ah', 'Qc' ],
    chips: 1000
};

let descriptors =
    Object.getOwnPropertyDescriptors( player );

console.log( descriptors );
console.log();
console.log( descriptors.cards );
```



`Object.getOwnPropertyDescriptors` returns all property descriptors in an object with the same keys as the keys of the original object. The following four property descriptors are returned (source: developer.mozilla.com):

- `value`: the value of the property
- `writable`: `true` if and only if the value associated with the property may be changed (data descriptors only)
- `get`: A function which serves as a getter for the property, or undefined if there is no getter (accessor descriptors only)
- `set`: A function which serves as a setter for the property, or undefined if there is no setter (accessor descriptors only)
- `configurable`: `true` if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object
- `enumerable`: `true` if and only if this property shows up during

enumeration of the properties on the corresponding object

Let's construct an example for getters and setters:

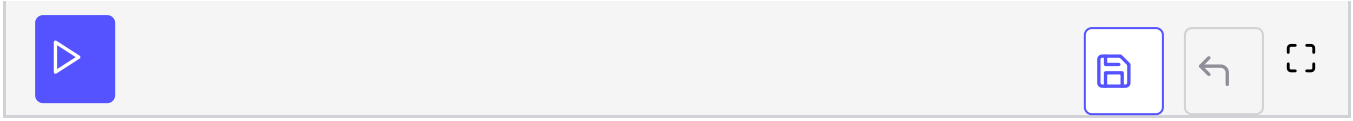
```
let player = {
    cards: [ 'Ah', 'Qc' ],
    chips: 1000,
    flop: [ '7d', '7c', '2c' ],
    get hand() {
        return [ ...this.cards, ...this.flop ];
    },
    set hand( newHand ) {
        if ( newHand.length && newHand.length === 5 ) {
            [ this.cards[0],
              this.cards[1],
              ...this.flop
            ] = newHand;
        }
    }
};

let descriptors =
    Object.getOwnPropertyDescriptors( player );

console.log(descriptors);
//> Object {
//    cards: Object,
//    chips: Object,
//    flop: Object,
//    hand: Object
//}

console.log( Object.keys( descriptors.hand ) );
//> ["get", "set", "enumerable", "configurable"]

console.log(descriptors.hand.get);
//> function get hand() {
//    return [ ...this.cards, ...this.flop ];
//  }
```

`Object.getOwnPropertyDescriptors` handles String keys as well as Symbol keys.

To show Symbol keys in node and some browsers, instead of `console.log`, use `console.dir` with the flag `showHidden: true`. Check out this node issue for more information, or re-read the relevant section of Chapter 9.

```
let s = Symbol('test');
let test = {
    [s]: 'test'
};
```

```
console.log( Object.getOwnPropertyDescriptors( test ) );
```



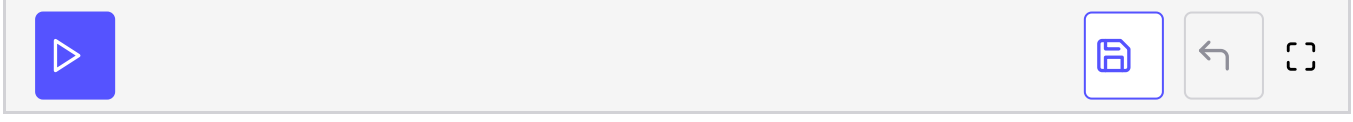
As a result, `Object.getOwnPropertyDescriptors` can be used to make shallow copies of objects using `Object.create`.

`Object.create` takes two arguments:

- the prototype of the object we wish to clone,
- the property descriptors of the object.

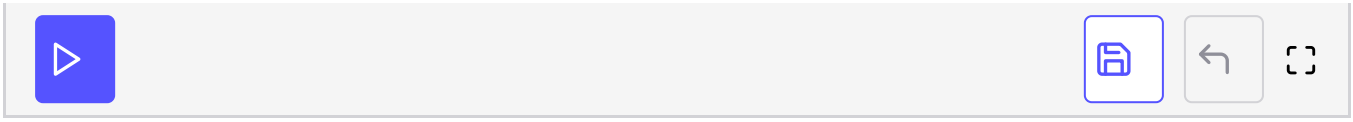
To illustrate the difference between a *shallow copy* and a *deep copy*, let's create a shallow copy of the `player` object defined above.

```
let player = {
    cards: [ 'Ah', 'Qc' ],
    chips: 1000,
    flop: [ '7d', '7c', '2c' ],
    get hand() {
        return [ ...this.cards, ...this.flop ];
    },
    set hand( newHand ) {
        if ( newHand.length && newHand.length === 5 ) {
            [ this.cards[0],
              this.cards[1],
              ...this.flop
            ] = newHand;
        }
    }
};

let proto = Object.getPrototypeOf( player );
let descriptors =
    Object.getOwnPropertyDescriptors( player );

let newPlayer = Object.create( proto, descriptors );

newPlayer.chips = 1500;

console.log( player.chips, newPlayer.chips );
```



We have created two seemingly independent entities. However, when trying to change a card of the new player, the change will be made in the context of the old player as well.

```
newPlayer.cards[1] = 'Ad';
console.log( newPlayer.cards[1], player.cards[1] );
```



This is because shallow copying only copied the reference of the cards array to the new player object. The original and the copied reference point to the same array.

Now, let's study some new array extensions.