Exercise on the Reflect API

In this exercise, we will use various methods of the Reflect API to create and alter objects.

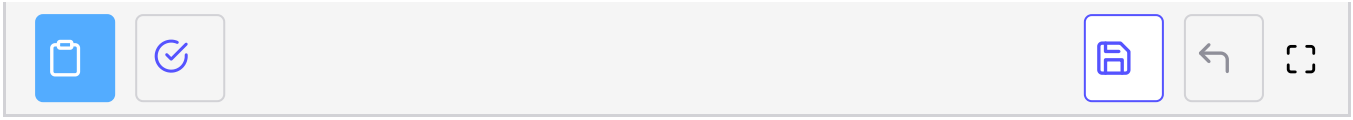
Exercise 1:

Create a `Movie` class, and initialize the `title` (`String`) and `movieLength` (`Number`) properties in the constructor. Create a `toString` method that prints the movie out using the following format:

```
${this.title} (${this.movieLength} minutes)
```

```
class Movie {
  constructor( title, movieLength ) {
    //Write your code here
  }

  toString() {
     return "Your Answer";
  }
};
```



Exercise 2:

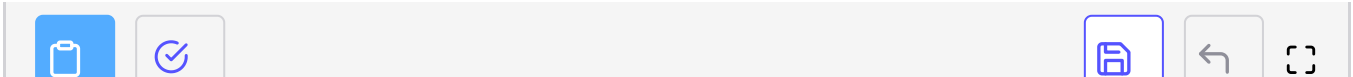
Use `Reflect.apply` to call the `toString` method of the `Movie` class with the following redefined properties: `"Rush"`, `123`.

```
//Assume class Movie already defined as in the preceeding solution

const rush123 = Reflect.apply(
  //Write your code here

  );

//Without your code, this is going to throw
//TypeError: Function.prototype.apply was called on undefined
```



Exercise 3:

Use the `Reflect` API to access a reference to the `Building` class assuming that initially you only have access to the `myBuilding` object. Then extend the prototype of the `Building` class by adding a `toString` method.

```
let myBuilding = (function() {
    class Building {
        constructor( address ) {
            this.address = address;
        }
    }

    class ResidentialBuilding extends Building {
        constructor( address, capacity ) {
            super( address );
            this.capacity = capacity;
        }
    }

    let myBuilding = new ResidentialBuilding(
        'Java Street 3',
        16
    );

    return myBuilding;
})();

let toString = function() {
    return `Address: ${this.address}`;
};

//Add your code here
```

Exercise 4:

Suppose a `Person` class is given.

```
class Person {
    constructor( name ) {
        this.name = name;
    }
    set name( name ) {
        let [ first, last ] = name.split(' ');
        this.first = first;
        this.last = last;
    }
}
```

Let's create a `person` object and a `newContext` variable.

```
let person = new Person( 'Julius Caesar' );
let newContext = { name: 'Marcus Aurelius' };
```

If we query the contents of a `person`, we can see how the setter transformed the name into a `first` and a `last` field.

```
person
//> Person {first: "Julius", last: "Caesar"}
```

Let's call a `Reflect.set` operation, setting the `name` field of our person object, and let's add the new context in the fourth variable.

```
Reflect.set(
    person,
    'name',
    'Alexander Severus',
    newContext
);
```

Determine the following values without executing the code:

- the return value of the above `Reflect.set` call
- `person.first` and `person.last`
- `person.name`
- `newContext.first` and `newContext.last`
- `newContext.name`

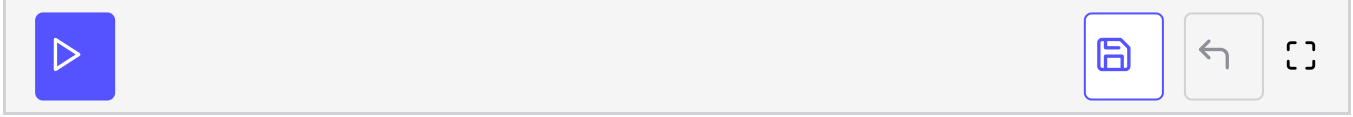
```
class Person {
    constructor( name ) {
        this.name = name;
    }
    set name( name ) {
        let [ first, last ] = name.split(' ');
        this.first = first;
        this.last = last;
    }
}

let person = new Person( 'Julius Caesar' );
let newContext = { name: 'Marcus Aurelius' };
```

```
const returnVal = Reflect.set(
    person,

    'name',
    'Alexander Severus',
    newContext
);

console.log(returnVal);
console.log(person.first + " " + person.last);
console.log(person.name);
console.log(newContext.first + " " + newContext.last);
console.log(newContext.name);
```

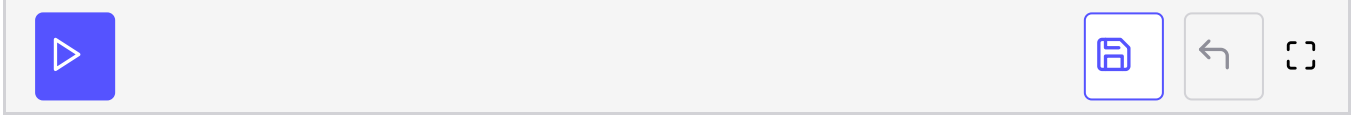


> **Pro Tip**
>
> propertyDescriptor.configurable is the value you should look for when judging whether an object property is configurable

Exercise 5:

```
let target = {};
let key = 'response';
let attributes = {
    value: 200,
    writable: true,
    enumerable: true
};

Reflect.defineProperty(
    target,
    key,
    attributes
);
```



Let's try to delete `target.response`.

```
console.log(Reflect.deleteProperty( target, key ));
//> false

console.log(target)
//> Object {response: 200}
```

The return value of the `deleteProperty` call indicates that the deletion is unsuccessful. Why? How can we modify the code such that the same `Reflect.deleteProperty` call returns `true`, and `target.response` is deleted?

```
let target = {};
let key = 'response';
let attributes = {
    value: 200,
    writable: true,
    enumerable: true
};

Reflect.defineProperty(
    target,
    key,
    attributes
);
```