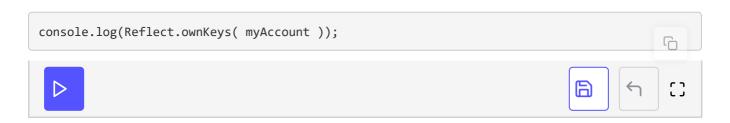
Property Access and Modification

access, modify (update and delete) properties of Objects

Reflect.has determines if a property exists for a given target object. The call enumerates all properties, not only own properties.

```
let target = class Account {
    constructor( name, email ) {
       this.name = name;
        this.email = email;
    get contact() {
        return `${this.name} <${this.email}>`;
    }
};
let args = [
    'Zsolt',
    'info@zsoltnagy.eu'
];
let myAccount = Reflect.construct(
    target,
    args );
console.log(Reflect.has( myAccount, 'name' ));
console.log(Reflect.has( myAccount, 'contact' ));
```

Reflect.ownKeys returns all own properties of a target in an array.



Reflect.get gets a property based on a key. As an optional parameter, the this context can be specified.

```
console.log(Reflect.get( myAccount, 'name' ));
//> "Zsolt"

console.log(Reflect.get( myAccount, 'contact' ));
//> "Zsolt - 555-1269"
```

Getting the name property of myAccount is straightforward. In the second example, getting the contact property requires the execution of a getter method. As we redefined this getter method using Reflect.setPrototypeOf not too long ago, the result becomes "Zsolt - 555-1269".

If we specify the context as the third argument of Reflect.get, we will get another result for the contact property:

```
console.log(Reflect.get(
    myAccount,
    'contact',
    { name: 'Bob' }
));
//> "Bob - 555-1269"
```

We can also set a property of our target using Reflect.set.

```
let target = myAccount;
let property = 'age';
let newValue = 32;

Reflect.set(
    myAccount,
    property,
    newValue
);

console.log(myAccount.age );
//> 32
```

The fourth argument of Reflect.set is an optional context. Solve exercise 4 to figure out how the context works in Reflect.set.

Reflect.defineProperty defines a new property. It is similar to calling Object.defineProperty. The difference between the two calls is that Reflect.defineProperty returns a boolean, while Object.defineProperty returns the object itself.

In the following example, we will define a writable property:

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

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

For a complete list of flags and their default values, check out the documentation of Object.defineProperty. Notice that all configuration flags inside the attributes object have a default value of false. This means that without specifying the writable flag, target.response would have been a read-only property.

We have accessed, created, and updated properties. The last operation missing is the deletion of properties.

```
let response = {
    status: 200
};

console.log(Reflect.deleteProperty( response, 'status' ));
//> true

console.log(response);
//> {}
```

Exercise 5 is about Reflect.deleteProperty and Reflect.defineProperty. You will find out that in some circumstances, deleting a property is not possible.

It is possible to prevent property extensions by calling the preventExtensions
method of the Reflect API:

```
let test = {
    title: 'Petri Nets',
    maxScore: 100
};

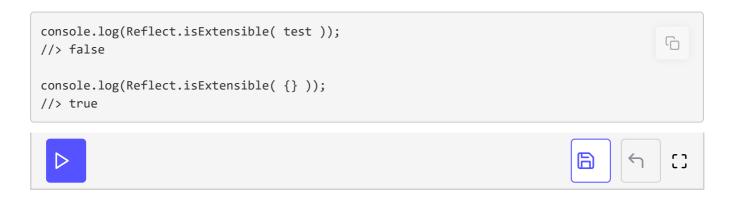
console.log(Reflect.preventExtensions( test ));
//> true

test.score = 55;

console.log( test );
//> Object {title: "Petri Nets", maxScore: 100}
```

As you can see, the score field is not added to the test object, as test was locked down by Reflect.preventExtensions.

Reflect.isExtensible tests whether an object is extensible:



Now, let's solve some exercises before learning new concepts.