Tip 40: Simplify Interfaces with get and set

In this tip, you'll learn how to mask logic behind simple interfaces with get and set.

We'll cover the following

Protecting class properties

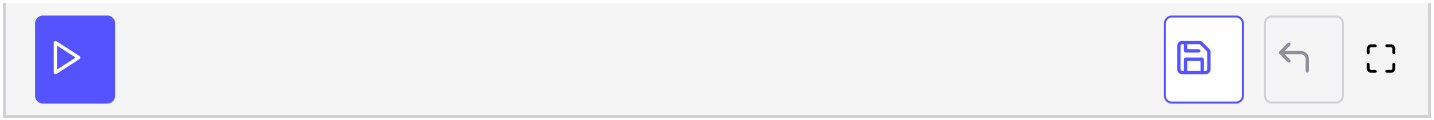
You have the basics of classes. You can *create instances, call properties, call methods, and extend parent classes*. But it won't be long before someone tries to alter a property you had no intention of exposing. Or maybe someone sets the wrong data type on a property, creating bugs because the code expects an integer, not a string.

One of the major problems in JavaScript is that there are no *private* properties by default. Everything is *exposed*. You can't control what the users of your class do with the methods or properties.

Think about your `Coupon`. It has a property of `price`, which you initially set in the constructor. A user of the class can access the property on an instance with *dot* syntax: `coupon.price`. So far, no problem. But because an instance of `Coupon` is just an object, the user can also change the property: `coupon.price = 11`.

In itself that's not a big deal. But you'll eventually hit a problem where another developer (or, admit it, you yourself) innocently tries to set a value other parts of the code may not expect. For example, what if instead of setting the `price` with an *integer*, you set it using a *string*? The change may seem harmless, but because all methods expect an integer, the change could ripple through the class, creating unexpected bugs.

```
class Coupon {
    constructor(price, expiration) {
        this.price = price;
        this.expiration = expiration || 'Two Weeks';
    }
    getPriceText() {
        return `${this.price}`;
    }
    getExpirationMessage() {
        return `This offer expires in ${this.expiration}`;
    }
}
const coupon = new Coupon(5);
coupon.price = '$10';
console.log(coupon.getPriceText());
```



When you set the `price` to a string, your message looks broken. *What can you do about it?*

Getters & setters

One solution is to put properties behind extra logic using *getters and setters*. A getter or setter is a way to *mask* complexity by making a function appear like a property.

The change is very simple. You already have a few functions that are clearly getting data. You have a `getPriceText()` method and a `getExpirationMessage()` method that have the word **"get"** built right in the function name. And, of course, to execute the method, you call it with *dot* syntax: `coupon.getPriceText()`.

Using `get`

Refactoring the method to a *getter* is simple. You simply add the keyword `get` in front of the method. After that, you can also rename the function to be a noun instead of an action. By convention, methods or functions are usually verbs and properties are usually nouns.

Here are your methods converted to getters. Notice the only change is the `get` keyword.

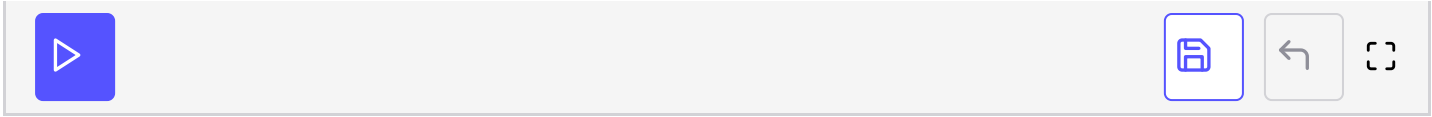
```
class Coupon {
    constructor(price, expiration) {
        this.price = price;
        this.expiration = expiration || 'two weeks';
```

```
    }
    get priceText() {
        return `${this.price}`;

    }
    get expirationMessage() {
        return `This offer expires in ${this.expiration}.`;
    }
}
```

After making that small change, you can call the method using *dot* syntax but *without* the parentheses. The method acts like a *property* even though you're executing code.

```
class Coupon {
    constructor(price, expiration) {
        this.price = price;
        this.expiration = expiration || 'two weeks';
    }
    get priceText() {
        return `${this.price}`;
    }
    get expirationMessage() {
        return `This offer expires in ${this.expiration}.`;
    }
}
const coupon = new Coupon(5);
coupon.price = 10;
console.log(coupon.priceText);
console.log(coupon.expirationMessage);
```



This makes information easier to retrieve, but it doesn't solve your problem of someone setting a bad value. To address that, you also need to create a setter.

Using `set`

A *setter* works like your *getter*. It *masks* a method by making the method appear like a *property*. A setter, though, accepts a *single* argument and changes a property rather than just exposing information. You don't pass the argument using parentheses. Instead, you pass the object using the equal sign (`=`) as if you were setting a value on an object.

As an example, make a setter that sets the `price` to half of an argument. This may not be a very useful setter, but it will show you how easy it is to mask method logic behind a setter.

To create a setter, you add the keyword `set` in front of a method. Inside the

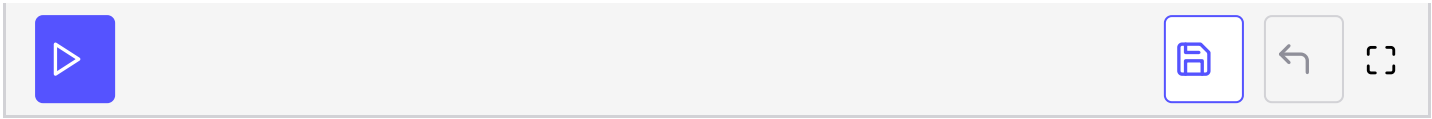
method, you can change a value on a property.

```
class Coupon {
    constructor(price, expiration) {
        this.price = price;
        this.expiration = expiration || 'Two Weeks';
    }
    set halfPrice(price) {
        this.price = price / 2;
    }
}
```

The problem with setters is that if you don't have a corresponding getter, things get a little odd. You can set a value to `halfPrice`. It looks like it's a normal property, but you can't get a value from `halfPrice`.

```
class Coupon {
    constructor(price, expiration) {
        this.price = price;
        this.expiration = expiration || 'Two Weeks';
    }
    set halfPrice(price) {
        this.price = price / 2;
    }
}

const coupon = new Coupon(5);
console.log(coupon.price);
coupon.halfPrice = 20;
console.log(coupon.price);
console.log(coupon.halfPrice);
```

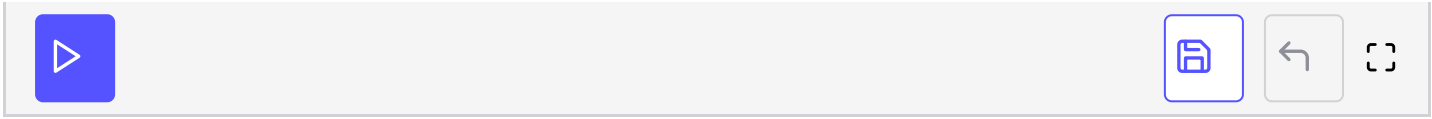


For this reason, it's always a good idea to pair getters and setters. In fact, they can (and should) have the same name. That's perfectly valid. What you can't do is have a property with the same name as your getter or setter. That would be invalid and create a lot of confusion.

For example, if you tried to make a setter of `price`, it would trigger an infinite call stack.

```
class Coupon {
    constructor(price, expiration) {
        this.price = price;
        this.expiration = expiration || 'Two Weeks';
    }
    get price() {
        return this.price;
    }
```

```
    set price(price) {
        this.price = `${price}`;
    }
}
const coupon = new Coupon(5);
// RangeError: Maximum call stack size exceeded
```



Using a bridge property

The solution is to use another property as a *bridge* between your getter and setter. You don't want users or other developers to access your *bridge property*. You want it to be for internal use only. In most languages, you'd use a *private* property. Because you don't have those currently in JavaScript, you must rely on convention.

Developers signal that a method or property is private by prepending it with an **underscore**. If you see an object with a property of `_price`, you should know you shouldn't access it directly.

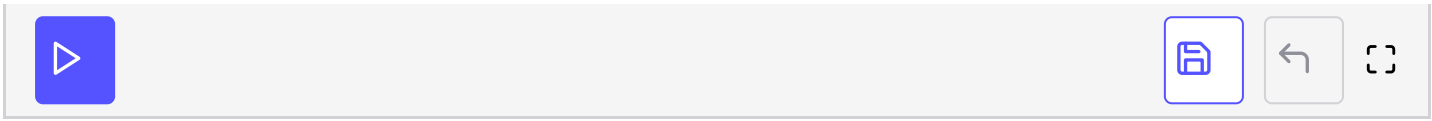
After you set an intermediate property, you can use getters and setters with the same name, *minus* the underscore, to access or update the value.

You now have the tools to solve your problem with setting a non-integer to `price`. Simply change the property `this.price` to `this._price` in the constructor. After that, create a getter to access `this._price` and a setter that will replace any non-numeric characters with nothing, leaving only the integers. This isn't perfect because it would strip out decimal points, but it's good for this demo.

```
class Coupon {
    constructor(price, expiration) {
        this._price = price;
        this.expiration = expiration || 'Two Weeks';
    }
    get priceText() {
        return `${this._price}`;
    }
    get price() {
        return this._price;
    }
    set price(price) {
        const newPrice = price
            .toString()
            .replace(/[^\d]/g, '');
        this._price = newPrice;
    }
    get expirationMessage() {
        return `This offer expires in ${this.expiration}`;
    }
```

```
}
const coupon = new Coupon(5);
console.log(coupon.price);

coupon.price = '$10';
console.log(coupon.price);
console.log(coupon.priceText);
```



A bonus to using this approach is you don't need to refactor any existing code. All code that currently uses `coupon.price` will work as intended.

The big advantage with getters and setters is that you hide complexity. The downside is that you mask your intentions. If another developer is writing code elsewhere, they may think they're setting a property when they're actually calling a method. Getters and setters can sometimes be very hard to debug and hard to test. As always, use with caution and make sure your intentions are clear with plenty of tests and documentation.

In the next tip, you'll learn another technique to mask complexity by turning data structures into iterables with generators.