Method Types

 (https://github.com/learn-co-curriculum/phase-1-method-types)  (https://github.com/learn-co-curriculum/phase-1-method-types/issues/new)

Learning Goals

- Recognize the syntactic differences between regular, static, getter and setter methods
- Recognize the different uses of each method type

Introduction

So far, we've seen some examples of classes that have their own custom methods:

```
class Square {
  constructor(sideLength) {
    this.sideLength = sideLength;
  }

  area() {
    return this.sideLength * this.sideLength;
  }
}
```

It turns out, however, there are four different types of methods we can write in a class: the standard 'instance' method we've seen already, *static*, getter and setter methods. Each of these behaves differently, and this variety provides us with flexibility in how we design the behaviors of our classes.

In this lesson, we're going to briefly look at each type of method and consider some use cases for each.

Standard Methods

Most class methods you will see use the following, standard syntax:

```
area() {
  return this.sideLength * this.sideLength;
}
```

These methods are available to any instance of the class they belong to, as we've seen:

```
const square = new Square(5);
square.area(); // => 25
```

Methods can be called from inside other methods just like properties:

```
class Square {
  constructor(sideLength) {
    this.sideLength = sideLength;
  }

  area() {
    return this.sideLength * this.sideLength;
  }

  areaMessage() {
    return `The area of this square is ${this.area()}`;
  }
}
const square = new Square(5);
square.area(); // => 25
square.areaMessage(); // => LOG: The area of this square is 25
```

In the class above, we can access `area()` directly, or use it to provide dynamic content for other methods. These methods are the most common - they act as the 'behaviors' of a class instance.

Static Methods

Static methods are class level methods - they are not callable on instances of a class, only the class itself. These are often used in 'utility' classes - classes that encapsulate a set of related methods but don't need to be made into instances. For example, we could write a `CommonMath` class that stores a series of math related methods:

```
class CommonMath {
  static triple(number) {
    return number * number * number;
  }

  static findHypotenuse(a, b) {
    return Math.sqrt(a * a + b * b);
  }
}
```

To access, these static methods:

```
const num = CommonMath.triple(3);
num; // => 27
const c = CommonMath.findHypotenuse(3, 4);
c; // => 5
```

This sort of class might be useful in many different situations, but we don't ever need an *instance* of it.

Define `get` Keyword in JavaScript Class Context

Often, when writing methods for a class, we want to return information derived from that instance's properties. In the `Square` class example earlier, `area()` returns a calculation based on `this.sideLength`, and `areaMessage()` returns a `String`.

In modern JavaScript, new syntax, `get`, has been introduced. The `get` keyword is used in classes for methods which serve the specific purpose of retrieving data from an instance.

The `get` keyword turns a method into a 'pseudo-property', that is - it allows us to write a method that interacts like a property. To use `get`, write a class method like normal, preceded by `get`:

```
class Square {
  constructor(sideLength) {
    this.sideLength = sideLength;
  }

  get area() {
    return this.sideLength * this.sideLength;
  }
}
```

As a result of this, `area` will now be available as though it is a property just like `sideLength`:

```
const square = new Square(5);
square.sideLength; // => 5
square.area; // => 25
```

If you try to use `this.area()`, you'll receive a TypeError - `area` is no longer considered a function!

This may seem strange - you could also just write the following and achieve the same result:

```
class Square {
  constructor(sideLength) {
    this.sideLength = sideLength;
    this.area = sideLength * sideLength;
  }
}
```

This is valid code, but what we've done is load our `constructor` with more calculations.

The main benefit to using `get` is that your `area` calculation isn't actually run until it is accessed. The 'cost' of calculating is offset, and may not be called at all. While our computers can make short work of this example, there are times when we need to perform calculations that are CPU intensive, sometimes referred to as a 'costly' or 'expensive' processes.

If included in the `constructor`, an expensive process will be called every time a new instance of a `class` is created. When dealing with many instances, this can result in decreased performance.

Using `get`, an expensive process can be delayed - only run when we need it, distributing the workload more evenly.

Even if your process is not expensive, using `get` is useful in general when deriving or calculating data from properties. Since properties can change, any values dependent on them should be calculated based on the current property values, otherwise we will run in to issues like this:

```
class Square {
  constructor(sideLength) {
    this.sideLength = sideLength;
    this.area = sideLength * sideLength;
  }
}
const square = new Square(5);
square.area; // => 25
square.sideLength = 10;
square.area; // => 25
```

If `area` is only calculated in the beginning and `sideLength` is then modified, `area` will no longer be accurate.

Define `set` Keyword in JavaScript Class Context

Using `get` to create a pseudo-property is only half the story, since it is only used for retrieving data from an instance. To change data, we have `set`.

The `set` keyword allows us to write a method that interacts like a property being assigned a value. By adding it in conjunction with a `get`, we can create a 'reassignable' pseudo-property.

For example, in the previous section we used `get` in the `Square` class:

```
class Square {
  constructor(sideLength) {
    this.sideLength = sideLength;
  }

  get area() {
    return this.sideLength * this.sideLength;
  }
}
```

This allowed us to retrieve the area of a Square instance like so:

```
const square = new Square(5);
square.sideLength; // => 5
square.area; // => 25
```

If we change `square.sideLength`, `square.area` will update accordingly:

```
square.sideLength = 10;
square.area; // => 100
```

However, we can't *assign* `area` a new value. To make `area` fully act like a real property, we create both `get` and `set` methods for it:

```
class Square {
  constructor(sideLength) {
    this.sideLength = sideLength;
  }

  get area() {
    return this.sideLength * this.sideLength;
  }
```

```
  set area(newArea) {
    this.sideLength = Math.sqrt(newArea);
  }
}
```

We can now 'set' the pseudo-property, `area`, and modify `this.sideLength` based on a reverse of the calculation we used in `get`:

```
const square = new Square(5);
square.sideLength; // => 5
square.area; // => 25
square.area = 64;
square.sideLength; // => 8
square.area; // => 64
```

We can now interact with `area` as though it is a modifiable property, even though `area` is derived.

From the outside, it looks like a property is being set, but behind the scenes, we can define what we want to happen, including applying conditional statements:

```
set area(newArea) {
  if (newArea > 0) {
    this.sideLength = Math.sqrt(newArea)
  } else {
    console.warn("Area cannot be less than 0");
  }
}
```

Creating pseudo-properties this way enables us to finely tune how data can be both accessed and modified. In using `get` and `set`, we are designing the interface for our class.

Using `get` and `set` with Private Fields

You may remember that in JavaScript, properties are public. That is, any property can be reassigned from outside. Here is where `get` and `set` really shine. With `get` and `set`, we can define the public facing methods for updating a private property:

```
class Square {
  #sideLength;
  constructor(sideLength) {
    this.#sideLength = sideLength;
  }

  get sideLength() {
    this.#sideLength;
  }

  set sideLength(sideLength) {
    this.#sideLength = sideLength;
  }
}
```

A square's side length must be a positive value. Now with our private field in place, we write code to make sure that `#sideLength` is always valid, both when an instance property is created and when it is modified:

```
class Square {
  #sideLength;
  constructor(sideLength) {
    if (sideLength > 0) {
      this.#sideLength = sideLength;
    } else {
      throw new Error("A square's side length must be a positive value");
    }
  }

  get sideLength() {
```

```
      this.#sideLength;
    }

    set sideLength(sideLength) {
      if (sideLength > 0) {
        this.#sideLength = sideLength;
      } else {
        throw new Error("A square's side length must be a positive value");
      }
    }
  }
```

We could always extract that duplicate code into a helper method, but the take away here is the design. We've designed our `Square` classes to be a little more resistant to unwanted changes that might introduce bugs.

Stepping away from `Squares` for a moment, let's consider an example with `String` properties. Imagine we want to build a `Student` class. The class takes in a students' first and last name. We are tasked with making sure names do not have any non-alphanumeric characters except for those that appear in names. This is sometimes referred to as *sanitizing* text.

With `set`, we can make sure that we sanitize input text both when an instance is created as well as later, if the property needs to change:

```
 class Student {
    #firstName;
    #lastName;

    constructor(firstName, lastName) {
      this.#firstName = this.sanitize(firstName);
      this.#lastName = this.sanitize(lastName);
    }

    get firstName() {
      return this.capitalize(this.#firstName);
    }
```

```
  set firstName(firstName) {
    this.#firstName = this.sanitize(firstName);
  }

  capitalize(string) {
    // capitalizes first letter
    return string.charAt(0).toUpperCase() + string.slice(1);
  }

  sanitize(string) {
    // removes any non alpha-numeric characters except dash and single quotes (apostrophes)
    return string.replace(/[^A-Za-z0-9-']+/g, "");
  }
}

let student = new Student("Carr@ol-Ann", ")Freel*ing");
student; // => Student { #firstName: 'Carrol-Ann', #lastName: 'Freeling' }

student.firstName = "Hea@)@(!$)ther";
student.firstName; // => 'Heather'
```

In this `Student` class, we've set up a pseudo-property, `firstName`, which refers to a private field `#firstName`. We've also included a `sanitize()` method that removes any non alpha-numeric characters except `-` and `'`.

Because we are using `set` and a private field, we can call `sanitize()` when a `Student` instance is constructed, *or* when we try to modify `#firstName`.

When to Use Methods Over `get` and `set`

Although `get` and `set` change the way in which we interact with a class, normal instance methods can do everything that `get` and `set` can do. So, which should we use and when? JavaScript itself is indifferent.

With `get` and `set`, while we don't gain any sort of extra functionality, we gain the ability to *differentiate* between behaviors. **We can use `get` and `set` whenever we are handling input or output of a class**. We are, in essence, creating the *public interface* of the class. We can treat this interface as a menu of sorts.. `get` and `set` methods are the ways in which *other* classes and code *should* utilize this class.

Using this design, all remaining methods can be considered *private*. They don't deal with input and output; they are only used internally as helper methods.

It is important to note that in JavaScript currently, we can *always* order off the menu. All class methods and properties are exposed for use 'publicly'. Using `get` and `set` in this way is purely design. In designing this way, however, we produce better organized, easier to understand classes.

Conclusion

In the Object Oriented JavaScript world, we have a variety of ways to build our classes. As we continue to learn about OO JS, we will see that this flexibility is important - it allows us to design many classes that work together, each serving their own specific purpose that we have defined.

Resources

- `get` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/get)
- `set` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/set)
- static methods (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/static)
- Working with private class features (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Working_With_Private_Class_Features)