Classes and Instances

[(https://github.com/learn-co-curriculum/phase-1-classes-and-instances)](https://github.com/learn-co-curriculum/phase-1-classes-and-instances) [(https://github.com/learn-co-curriculum/phase-1-classes-and-instances/issues/new)](https://github.com/learn-co-curriculum/phase-1-classes-and-instances/issues/new)

Learning Goals

- Identify the creation of `class` instances using `constructor`
- State the definition of instance properties

Introduction

In Object-Oriented JavaScript, objects share a similar structure, the `class` . Each `class` has the ability to generate copies of itself, referred to as *instances*. Each of these `class` instances can contain unique data, often set when the instance is created.

In this lesson, we are going to take a closer look at `class` syntax, instance creation and how to use the `constructor` .

A Basic `class`

The `class` syntax was introduced in **ECMAScript 2015** [(https://www.w3schools.com/js/js_es6.asp)](https://www.w3schools.com/js/js_es6.asp) and it's important to note that the `class` keyword is just syntactic sugar, or a nice abstraction, over JavaScript's existing prototypal object structure.

> Reminder: All JavaScript objects inherit properties and methods from a `prototype` . This includes standard objects like functions and data types.

A basic, empty class can be written on one line:

```
class Fish {}
```

With only a name and brackets, we can now create instances of the 'Fish' `class` by using `new` :

```
const oneFish = new Fish();
const twoFish = new Fish();

oneFish; // => Fish {}
twoFish; // => Fish {}

oneFish == twoFish; // => false
```

These two fish are unique `class` instances, even though they have no information encapsulated within them.

Using the `constructor`

Typically, when we create an instance of a `class`, we want it to contain some bit of unique information from the beginning. To do this, we use a special method called `constructor`:

```
class Fish {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
}
```

The `constructor` method allows us pass arguments in when we use the `new` syntax:

```
const redFish = new Fish("Red", 3);
const blueFish = new Fish("Blue", 1);

redFish; // => Fish { name: 'Red', age: 3 }
blueFish; // => Fish { name: 'Blue', age: 1 }
```

Now our instances are each carrying unique data. It is possible to add and change data using other means *after* an instance is created using custom methods, but the `constructor` is where any initial data is defined.

Assigning Instance Properties

We see that our fish have data, but what is happening exactly inside the `constructor` ?

```
constructor(name, age) {
  this.name = name;
  this.age = age
}
```

Two arguments, `name` and `age` are passed in and then assigned to something new: `this` .

For now, think of `this` as a reference to the object it is inside. Since we're calling `constructor` when we create a new instance (`new Fish('Red', 3)`), `this` is referring to the *instance we've created*. *This* fish.

> In `class` methods, `this` can be used to refer to properties of an instance, like `name` and `age` , or methods of an instance (`this.sayName()`). There is more to `this` than meets the eye, however, and we will go into more detail later on.

Accessing Instance Properties

If we've assigned an instance to a variable, we can access properties using the variable object:

```
const oldFish = new Fish("George", 19);
const newFish = new Fish("Clyde", 1);

oldFish.name; //=> 'George'
oldFish.age; //=> 19
newFish.name; //=> 'Clyde'
newFish.age; //=> 1
```

By using `this.name` and `this.age` to define properties in our `constructor` , we can also refer to these properties within other methods of our `class` :

```
class Fish {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayName() {
    return `Hi my name is ${this.name}`;
  }
}
```

This allows us to return dynamic information based on the unique properties we assigned back when an instance was created. Another example:

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

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

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

Private Properties

All properties are accessible from outside an instance, as we see with `square.sideLength` , as well as from within `class` methods (`this.sideLength`).

This is not always desirable - sometimes, we want to protect the data from being modified after being set, or we want to use methods to control the exact ways our data should be changed. Say, for instance, we had a `Transaction` `class` that we are using to represent individual bank transactions. When a new `Transaction` instance is created, it has `amount`, `date` and `memo` properties.

```
class Transaction {
  constructor(amount, date, memo) {
    this.amount = amount;
    this.date = date;
    this.memo = memo;
  }
}
```

The `date`, `amount` and `memo` properties represent fixed values for each instance when a `Transaction` instance is created and probably shouldn't be altered. However, it is still possible to change these properties after they are assigned:

```
const transaction = new Transaction(100.24, "03/04/2018", "Grocery Shopping");
transaction.amount; // => 100.24
transaction.amount = 1000000000000.24;
transaction.amount; // => 1000000000000.24
```

Historically, JavaScript has not provided any way to make a property private - all `class` and object properties were exposed as we see above. The only option available was to follow a common convention, used by many JavaScript programmers to indicate properties that are not intended to be accessed from outside the `class`:

```
class Transaction {
  constructor(amount, date, memo) {
    this._amount = amount;
    this._date = date;
    this._memo = memo;
  }
}
```

In the code above, you'll see that an underscore (`_`) has been prepended to the name of each property. This has no effect on how the code functions - it simply indicates to other programmers that that property or variable is intended to be private.

Recently, however, the ability to create private properties and methods in JavaScript classes has been added. A private field is created by prefixing its name with `#` . For this to work, the fields must first be declared at the top of the class definition. After declaring the fields, you can access them and assign values in methods within your class:

```
class Transaction {
  // declare private fields
  #amount;
  #date;
  #memo;
  constructor(amount, date, memo) {
    // assign values to private fields
    this.#amount = amount;
    this.#date = date;
    this.#memo = memo;
  }
}
```

If you try to assign values to the private properties in the constructor without declaring them first, you will get a syntax error.

Private elements declared using the `#` syntax cannot be accessed or changed from outside the class:

```
const transaction = new Transaction(100.24, "03/04/2018", "Grocery Shopping");
transaction.amount;
// => undefined
transaction.#amount;
// => SyntaxError
```

While **private class features (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields)** are relatively new in JavaScript, they are widely supported by all major browsers.

Conclusion

So, to recap, we can define a `class` simply by writing `class` , a name, and a set of curly brackets. We can then use this `class` to create unique instances. These instances can contain their own data, which we typically set using `constructor` , passing in arguments and assigning them to properties we've defined. With these properties, `class` instances can carry data around with them wherever they go. While there are no private properties (yet), it is possible to set up `class` es to emphasize using methods over directly changing properties.

Resources

- **Classes (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes)**
- **Sitepoint: Private Class Fields (https://www.sitepoint.com/javascript-private-class-fields/)**
- **MDN: Private Class Features (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields)**