Day 5: Inheritance

Classes in JavaScript

Functional Classes

In this section, we'll discuss some of the ways we can use functions to simulate the behavior of classes.

Using Functions

1. Define a normal JavaScript function.
2. Create an object by using the `new` keyword.
3. Define properties and methods for a created object using the `this` keyword.

EXAMPLE

```
'use strict';

function Fruit (type) {
    this.type = type;
    this.color = 'unknown';
    this.getInformation = getFruitInformation;
}

function getFruitInformation() {
    return 'This ' + this.type + ' is ' + this.color + '.';
}

let lime = new Fruit('Mexican lime');
console.log(lime.getInformation());

lime.color = 'green';
```

Output

Run

We can also define the *getInformation* function internally:

```
'use strict';

function Fruit (type) {
    this.type = type;
    this.color = 'unknown';
    this.getInformation = function() {
        return 'This ' + this.type + ' is ' + this.color + '.';
    }
}
```

Go to Top

```
10
11 let lime = new Fruit('Mexican lime');
12 console.log(lime.getInformation());
13
14 lime.color = 'green';
```

Output

Run

The Prototype Property

The drawback of internally defining the *getInformation* function is that it recreates that function every time we create a new *Fruit* object. Fortunately, every function in JavaScript has something called a *prototype property*, which is empty by default. We can think of a function's prototype as an object blueprint or paradigm; when we add methods and properties to the prototype, they are accessible to *all* instances of that function. This is especially useful for *inheritance* (discussed below).

We can add a function to our constructor function's prototype like so:

- EXAMPLE

```
1  'use strict';
2
3  function Fruit (type) {
4      this.type = type;
5      this.color = 'unknown';
6  }
7
8  Fruit.prototype.getInformation = function() {
9      return 'This ' + this.type + ' is ' + this.color + '.';
10 }
11
12 let lime = new Fruit('Mexican lime');
13 console.log(lime.getInformation());
14
15 lime.color = 'green';
```

Output

Run

Using Object Literals

We can use object literals to define an object's properties and functions by initializing a variable with a comma-separated list of property-value pairs enclosed in curly braces.

- EXAMPLE

```
1  'use strict';
2
3  let lime = {
4      type: 'Mexican lime',
5      color: 'green',
6      getInformation: function() {
7          return 'This ' + this.type + ' is ' + this.color + '.';
8      }
9  }
10
11 console.log(lime.getInformation());
12
13 lime.color = 'yellow';
```

Output

Run

Table Of Contents

Go to Top

Singleton Class Using a Function

A *singleton* class is a design pattern that restricts a class to a *single instance*. When we assign the value of `new function(){...}` to a variable, the following happens:

1. We define an anonymous constructor function.
2. We invoke the anonymous constructor function with the `new` keyword.

- EXAMPLE

```
'use strict';

let lime = new function() {
    this.type = 'Mexican lime';
    this.color = 'green';
    this.getInformation = function() {
        return 'This ' + this.type + ' is ' + this.color + '.';
    };
}

console.log(lime.getInformation());

lime.color = 'yellow';
```

Output

Run

Classes

JavaScript classes, introduced in *ECMAScript 6*, are essentially syntactic sugar over JavaScript's existing prototype-based inheritance that don't actually introduce a new object-oriented inheritance model. This syntax is a means of more simply and clearly creating objects and deal with inheritance.

We define classes in two ways:

Class Declarations

One way to define a class is using a class declaration. To declare a class, we use the `class` keyword and follow it with the class' name. Ideally, we always write class names in TitleCase.

- EXAMPLE

```
class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}

let p = new Polygon(1, 2);
```

Output

Run

An important difference between function declarations and class declarations is that function declarations are hoisted (i.e., can be referenced before they're declared) but class declarations are not. This means we must first declare our class before attempting to access (or reference) it; if we fail to do so, our code throws a *ReferenceError*.

Table Of Contents

Go to Top

- EXAMPLE

```
try {
    let p = new Polygon(1, 2);
    console.log('Polygon p:', p);
}
catch (exception) {
    console.log(exception.name + ': ' + exception.message);
}
class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
}

p = new Polygon(1, 2);
```

Output

Run

Class Expressions

Class expressions are another way to define a class, and they can be either *named* or *unnamed*. The name given to a named class expression is local to the class' body.

- EXAMPLE

Unnamed Class Expression

```
let Polygon = class {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
};

console.log('Polygon:', Polygon);
let p = new Polygon(1, 2);
```

Output

Run

Named Class Expression

```
let Polygon = class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
};

console.log('Polygon:', Polygon);
let p = new Polygon(1, 2);
```

Output

Run

The Constructor Method

Table Of Contents

Go to Top

- The *constructor method* is a special method we use to create and initialize objects of a class.
- A class can only have *one* special method with the name `constructor`, and attempting to write a class containing more than one constructor method will throw a *SyntaxError*.
- To implement *inheritance*, we can use the `super` keyword in a constructor to call a parent class constructor.

Prototype Methods

- EXAMPLE

```
'use strict';

class Polygon {
    constructor(height, width) {
        this.height = height;
        this.width = width;
    }
    getArea() {
        return this.height * this.width;
    }
}

const square = new Polygon(10, 10);

```

Output

Run

Static Methods

Static methods are methods relevant to all instances of a class — not just any one instance. These methods receive information from their arguments and not a class instance, which allows us to invoke a class' static methods without creating an instance of the class. In fact, we actually *can't* call a static method on an instantiated class object (attempting to do so throws a *TypeError*).

We define a class' static methods using the `static` keyword. We typically use these methods to create utility functions for applications, as they can't be called on class objects.

- EXAMPLE

```
'use strict';

class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }
    static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;
        return Math.sqrt(dx * dx + dy * dy);
    }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);

// The correct way to call a static method
console.log(Point.distance(p1, p2));
```

Table Of Contents

Go to Top

```
// Attempt to call a static method on an instance of the class
try {
    console.log(p1.distance(p1, p2));
}
catch (exception) {
    console.log(exception.name + ': ' + exception.message);
```

Output

Run

Inheritance

In essence, this construct allows us to create an object prototype or class that's an *extension* of another object prototype or class. A class inheriting from some other class (referred to as a superclass or parent class) is called a subclass (or child class). The subclass inherits the superclass' methods and behaviors, but it can also declare new ones or even override existing ones.

- INHERITANCE IN JAVA — Recommended Article

Inheritance in Java — View

Subclassing with the `extends` Keyword

We use the `extends` keyword in class declarations or class expressions to create a child class (i.e., subclass).

- EXAMPLE

```
'use strict';

class Animal {
    constructor(name) {
        this.name = name;
    }
    speak() {
        console.log(this.name, 'speaks.');
    }
}

class Dog extends Animal {
    speak() {
        console.log(this.name, 'barks.');
    }
}

let spot = new Dog('Spot');
spot.speak();

spot = new Animal('Spot');
```

Output

Run

We can also *extend* functional classes:

```
'use strict';

function Animal(name) {
    this.name = name;
}
```

Table Of Contents

Go to Top

```
6
7 Animal.prototype.speak = function() {
8     console.log(this.name, 'speaks.');
9 }
10
11 class Dog extends Animal {
12     speak() {
13         console.log(this.name, 'barks.');
14     }
15 }
16
17 let spot = new Dog('Spot');
18 spot.speak();
19
20 spot = new Animal('Spot');
```

Output

Run

Superclass Calls Using the `super` Keyword

We use the *super* keyword to call functions on an object's parent.

\- EXAMPLE

```
1  'use strict';
2
3  class Animal {
4      constructor(name) {
5          this.name = name;
6      }
7      speak() {
8          console.log(this.name, 'speaks.');
9      }
10 }
11
12 class Dog extends Animal {
13     speak() {
14         super.speak();
15         console.log(this.name, 'barks.');
16     }
17 }
18
19 let spot = new Dog('Spot');
```

Output

Run

Extending an Object

The ability to extend multiple classes from the same superclass (or model multiple object types after the same prototype) is powerful because it provides us with certain implied guarantees about the basic functionality of the subclasses; as extensions of the parent class, subclasses are guaranteed to (at minimum) have the superclass' fields, methods, and functions.

\- EXAMPLE

In this example, we call the superclass constructor using `super()`, override a superclass function (`speak()`), add an additional property (`collarColor`), and add a new subclass method (`collar()`).

Table Of Contents

Go to Top

```
'use strict';

class Animal {
    constructor(name) {
        this.animalType = 'Animal'
        this.name = name;
    }
    type() {
        console.log(this.name, 'is type', this.animalType);
    }
    speak() {
        console.log(this.name, 'speaks.');
    }
}

class Dog extends Animal {
    constructor(name, collarColor) {
        super(name);
        this.animalType = 'Dog';
        this.collarColor = collarColor;
    }
    speak() {
        console.log(this.name, 'barks.');
    }
    collar() {
        console.log(this.name, 'has a', this.collarColor, 'collar.')
    }
}

let spot = new Dog('Spot', 'red');
spot.type();
spot.speak();
spot.collar();

// Because the Animal constructor only expects one argument,
// only the first value passed to it is used
spot = new Animal('Spot', 'white');
spot.type();
spot.speak();
try {
    spot.collar();
}
catch (exception) {
    console.log(exception.name + ': ' + exception.message
    + ' (collar is a method of Dog, not Animal).');
}
```

Output

Run

Related challenge for **Classes in JavaScript**

Day 4: Classes

Success Rate: 99.65% Max Score: 15 Difficulty:

Solve Challenge

Table Of Contents