Context Binding

introduction to context binding, animation of a ball using the setInterval method

In ES5, function scope often requires us to bind the context to a function. Context binding is usually performed in one of the following two ways:

1. by defining a `self = this` variable,
2. by using the `bind` function.

In our first example, we will attempt to animate a ball using the `setInterval` method.

```
var Ball = function( x, y, vx, vy ) {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.dt = 25; // 1000/25 = 40 frames per second
    setInterval( function() {
        this.x += vx;
        this.y += vy;
        console.log( this.x, this.y );
    }, this.dt );

}

var ball = new Ball( 0, 0, 10000, 10000 );
```

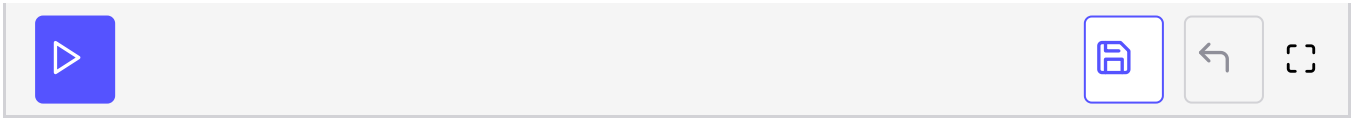
The code times out because the setInterval method runs an infinite number of times.

The animation failed, because inside the function argument of `setInterval`, based on the rule of function scoping, the value of `this` is different.

To access and modify the variables in the scope of the `ball` object, we have to make the context of the ball accessible inside the function argument. Our first solution looks like this:

```
var Ball = function( x, y, vx, vy ) {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.dt = 25; // 1000/25 = 40 frames per second
    var self = this;
    setInterval( function() {
        self.x += vx;
        self.y += vy;
        console.log( self.x, self.y );
    }, this.dt );
}

var ball = new Ball( 0, 0, 1, 2 );
```



The code times out because the setInterval method runs an infinite number of times.

This solution is still a bit awkward, as we have to maintain the `self` and `this` references. It is easy to make a mistake and use `this` instead of `self` somewhere in your code. Therefore, in ES5, best practices suggest using the `bind` method:

```
var Ball = function( x, y, vx, vy ) {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.dt = 25; // 1000/25 = 40 frames per second
    setInterval( function() {
        this.x += vx;
        this.y += vy;
        console.log( this.x, this.y );
    }.bind( this ), this.dt );
}
```

The `bind` method binds the context of the `setInterval` function argument to `this`.

In ES6, arrow functions come with automatic context binding. The lexical value of `this` isn't shadowed by the scope of the arrow function. Therefore, you save yourself thinking about context binding.

Let's rewrite the above example in ES6:

```
var Ball = function( x, y, vx, vy ) {
```

```
var Ball = function( x, y, vx, vy ) {
    this.x = x;
    this.y = y;
    this.vx = vx;
    this.vy = vy;
    this.dt = 25; // 1000/25 = 40 frames per second
    setInterval( () => {
        this.x += vx;
        this.y += vy;
        console.log( this.x, this.y );
    }, this.dt );
}

b = new Ball( 0, 0, 1, 1 );
```

The code times out because the setInterval method runs an infinite number of times.

> Use case: Whenever you want to use the lexical value of `this` coming from outside the scope of the function, use arrow functions.

Don't forget that the equivalence transformation for fat arrows is the following:

```
// ES2015
( ARGUMENTS ) => VALUE;

// ES5
function( ARGUMENTS ) { return VALUE; }.bind( this );
```

The same holds for blocks:

```
// ES2015
( ARGUMENTS ) => {
    // ...
};

// ES5
function( ARGUMENTS ) {
    // ...
}.bind( this );
```

In constructor functions and prototype extensions, it does not make sense to use fat arrows. This is why we kept the `Ball` constructor a regular function.

We will introduce the `class` syntax later to provide an alternative for

construction functions and prototype extensions.