Arrow Functions

Often times we have nested functions in which we would like to preserve the context of `this` from its lexical scope. An example is shown below:

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

Person.prototype.prefixName = function (arr) {
    return arr.map(function (character) {
        return this.name + character; // Cannot read property 'name' of undefined
    });
};
```

One common solution to this problem is to store the context of `this` using a variable:

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

Person.prototype.prefixName = function (arr) {
    var that = this; // Store the context of this
    return arr.map(function (character) {
        return that.name + character;
    });
};
```

We can also pass in the proper context of `this`:

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

Person.prototype.prefixName = function (arr) {
    return arr.map(function (character) {
        return this.name + character;
    }, this);
};
```

As well as bind the context:

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

Person.prototype.prefixName = function (arr) {
    return arr.map(function (character) {
        return this.name + character;
    }.bind(this));
};
```

Using **Arrow Functions**, the lexical value of `this` isn't shadowed and we can re-write the above as shown:

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

Person.prototype.prefixName = function (arr) {
    return arr.map(character => this.name + character);
};
```

> **Best Practice**: Use **Arrow Functions** whenever you need to preserve the lexical value of `this`.

Arrow Functions are also more concise when used in function expressions which simply return a value:

```
var squares = arr.map(function (x) { return x * x }); // Function Expression
```

```
const arr = [1, 2, 3, 4, 5];
const squares = arr.map(x => x * x); // Arrow Function for terser implementation
```

> **Best Practice**: Use **Arrow Functions** in place of function expressions when possible.