ARROW FUNCTION

An **arrow function expression** has a shorter syntax than a function expression and does not have its own this, arguments, super, or new.target. These function expressions are best suited for non-method functions, and they cannot be used as constructors.

```
var materials = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];
console.log(materials.map(material => material.length));
// expected output: Array [8, 6, 7, 9]
```

There are two benefits to arrow functions.

First, they are less verbose than traditional function expressions:

```
const arr = [1, 2, 3];
const squares = arr.map(x => x * x);

// Traditional function expression:
const squares = arr.map(function (x) { return x * x });
```

Second, their this is picked up from surroundings (lexical). Therefore, you don't need bind() or that = this, anymore.

```
function UiComponent() {
    const button = document.getElementById('myButton');
    button.addEventListener('click', () => {
        console.log('CLICK');
        this.handleClick(); // lexical `this`
    });
}
```

The following variables are all lexical inside arrow functions:

- arguments
- super
- this
- new.target

TRADITIONAL FUNCTIONS ARE BAD NON-METHOD FUNCTIONS, DUE TO `THIS`

In JavaScript, traditional functions can be used as:

1. Non-method functions
2. Methods
3. Constructors

These roles clash: Due to roles 2 and 3, functions always have their own `this`. But that prevents you from accessing the `this` of, e.g., a surrounding method from inside a callback (role 1).

You can see that in the following ES5 code:

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) { // (A)
    'use strict';
    return arr.map(function (x) { // (B)
        // Doesn’t work:
        return this.prefix + x; // (C)
    });
};
```

In line C, we’d like to access `this.prefix`, but can’t, because the `this` of the function from line B shadows the `this` of the method from line A. In strict mode, `this` is `undefined` in non-method functions, which is why we get an error if we use `Prefixer`:

```
> var pre = new Prefixer('Hi ');
> pre.prefixArray(['Joe', 'Alex'])
TypeError: Cannot read property 'prefix' of undefined
```

There are three ways to work around this problem in ECMAScript 5.

SOLUTION 1: `THAT = THIS`

You can assign `this` to a variable that isn’t shadowed. That’s what’s done in line A, below:

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
    var that = this; // (A)
    return arr.map(function (x) {
        return that.prefix + x;
    });
};
```

Now `Prefixer` works as expected:

```
> var pre = new Prefixer('Hi ');
> pre.prefixArray(['Joe', 'Alex'])
[ 'Hi Joe', 'Hi Alex' ]
```

SOLUTION 2: SPECIFYING A VALUE FOR `THIS`

A few Array methods have an extra parameter for specifying the value that `this` should have when invoking the callback. That's the last parameter in line A, below.

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
    return arr.map(function (x) {
        return this.prefix + x;
    }, this); // (A)
};
```

OLUTION 3: `BIND(THIS)`

You can use the method `bind()` to convert a function whose `this` is determined by how it is called (via `call()`, a function call, a method call, etc.) to a function whose `this` is always the same fixed value. That's what we are doing in line A, below.

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
Prefixer.prototype.prefixArray = function (arr) {
    return arr.map(function (x) {
        return this.prefix + x;
    }.bind(this)); // (A)
};
```

ECMASCRIPT 6 SOLUTION: ARROW FUNCTIONS

Arrow functions work much like solution 3. However, it's best to think of them as a new kind of functions that don't lexically shadow `this`. That is, they are different from normal functions (you could even say that they do less). They are not normal functions plus binding.

With an arrow function, the code looks as follows.

```
function Prefixer(prefix) {
    this.prefix = prefix;
}
```

```
Prefixer.prototype.prefixArray = function (arr) {
    return arr.map((x) => {
        return this.prefix + x;
    });
};
```

To fully ES6-ify the code, you'd use a class and a more compact variant of arrow functions:

```
class Prefixer {
    constructor(prefix) {
        this.prefix = prefix;
    }
    prefixArray(arr) {
        return arr.map(x => this.prefix + x); // (A)
    }
}
```

In line A we save a few characters by tweaking two parts of the arrow function:

- If there is only one parameter and that parameter is an identifier then the parentheses can be omitted.
- An expression following the arrow leads to that expression being returned.

`THIS` AND ARROW FUNCTIONS

In JavaScript, `this` is a variable that's set when a function is called. This makes it a very powerful and flexible feature, but it comes at the cost of always having to know about the context that a function is executing in. This is notoriously confusing, especially when returning a function or passing a function as an argument.

Let's look at an example:

```
let deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    createCardPicker: function() {
        return function() {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();
```

```
        alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Notice that `createCardPicker` is a function that itself returns a function. If we tried to run the example, we would get an error instead of the expected alert box. This is because the `this` being used in the function created by `createCardPicker` will be set to `window` instead of our `deck` object. That's because we call `cardPicker()` on its own. A top-level non-method syntax call like this will use `window` for `this`. (Note: under strict mode, `this` will be `undefined` rather than `window`).

We can fix this by making sure the function is bound to the correct `this` before we return the function to be used later. This way, regardless of how it's later used, it will still be able to see the original `deck` object. To do this, we change the function expression to use the ECMAScript 6 arrow syntax. Arrow functions capture the `this` where the function is created rather than where it is invoked:

```
        let deck = {
            suits: ["hearts", "spades", "clubs", "diamonds"],
            cards: Array(52),
            createCardPicker: function() {
                // NOTE: the line below is now an arrow function, allowing us to
        capture 'this' right here
                return () => {
                    let pickedCard = Math.floor(Math.random() * 52);
                    let pickedSuit = Math.floor(pickedCard / 13);

                    return {suit: this.suits[pickedSuit], card: pickedCard % 13};
                }
            }
        }

        let cardPicker = deck.createCardPicker();
        let pickedCard = cardPicker();

        alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Even better, TypeScript will warn you when you make this mistake if you pass the `--noImplicitThis` flag to the compiler. It will point out that `this` in `this.suits[pickedSuit]` is of type `any`

ARROW FUNCTION SYNTAX

The "fat" arrow => (as opposed to the thin arrow ->) was chosen to be compatible with CoffeeScript, whose fat arrow functions are very similar.

Specifying parameters:

```
            () => { ... } // no parameter
             x => { ... } // one parameter, an identifier
```

```
(x, y) => { ... } // several parameters
```

Specifying a body:

```
x => { return x * x }      // block
x => x * x                 // expression, equivalent to previous line
```

The statement block behaves like a normal function body. For example, you need `return` to give back a value. With an expression body, the expression is always implicitly returned.

Note how much an arrow function with an expression body can reduce verbosity. Compare:

```
const squares = [1, 2, 3].map(function (x) { return x * x });
const squares = [1, 2, 3].map(x => x * x);
```

NO LINE BREAK AFTER ARROW FUNCTION PARAMETERS

ES6 forbids a line break between the parameter definitions and the arrow of an arrow function:

```
const func1 = (x, y) // SyntaxError
=> {
    return x + y;
};
const func2 = (x, y) => // OK
{
    return x + y;
};
const func3 = (x, y) => { // OK
    return x + y;
};

const func4 = (x, y) // SyntaxError
=> x + y;
const func5 = (x, y) => // OK
x + y;
```

Line breaks INSIDE parameter definitions are OK:

```
const func6 = ( // OK
    x,
    y
) => {
    return x + y;
};
```

The rationale for this restriction is that it keeps the options open w.r.t. “headless” arrow functions in the future (you’d be able to omit the parentheses when defining an arrow function with zero parameters).