ARROW FUNCTIONS IN JAVASCRIPT

WHAT ARE ARROW FUNCTIONS?

- An **arrow function expression** is a syntactically compact alternative to a regular function expression, although without its own bindings to the this, arguments, super, or new.target keywords.
- Arrow function expressions are ill suited as methods, and they cannot be used as constructors.
- Arrow function is similar to lambda functions in java 8

SYNTAX

BASIC SYNTAX

(param1, param2, …, paramN) => { statements }

(param1, param2, …, paramN) => expression

// equivalent to: => { return expression; }

// Parentheses are optional when there's only one parameter name:

(singleParam) => { statements }

singleParam => { statements }

// The parameter list for a function with no parameters should be written with a pair of parentheses. () => { statements }

ADVANCED SYNTAX

- Parenthesize the body of a function to return an object literal expression:

```
params => ({foo: bar})
```

- Rest parameters and default parameters are supported

```
(param1, param2, ...rest) => { statements }
(param1 = defaultValue1, param2, …, paramN = defaultValueN) => { statements }
```

- Destructuring within the parameter list is also supported

```
var f = ([a, b] = [1, 2], {x: c} = {x: a + b}) => a + b + c; f(); // 6
```

WHY WE NEED ARROW FUNCTION?

- Two factors influenced the introduction of arrow functions
  - the need for shorter functions.
  - the behavior of "this" keyword.

SHORTER FUNCTIONS

```
var elements = [ 'Hydrogen', 'Helium', 'Lithium', 'Beryllium' ];
```

- This statement returns the array: [8, 6, 7, 9]

```
elements.map(function(element) {
        return element.length;
        }
);
```

- The regular function above can be written as the arrow function below

```
elements.map((element) => {
        return element.length;
}
); // [8, 6, 7, 9]
```

- When there is only one parameter, we can remove the surrounding parentheses

```
elements.map(element => {
        return element.length;
 }); // [8, 6, 7, 9]
```

- When the only statement in an arrow function is `return`, we can remove `return` and remove the surrounding curly brackets

```
elements.map(element => element.length); // [8, 6, 7, 9]
```

- In this case, because we only need the length property, we can use destructuring parameter. Notice that the `length` corresponds to the property we want to get whereas the obviously non-special `lengthFooBArX` is just the name of a variable which can be changed to any valid variable name you want

```
elements.map(({ length: lengthFooBArX }) => lengthFooBArX); // [8, 6, 7, 9]
```

- This destructuring parameter assignment can also be written as seen below. However, note that in this example we are not assigning `length` value to the made up property. Instead, the literal name // itself of the variable `length` is used as the property we want to retrieve from the object.

```
elements.map(({ length }) => length); // [8, 6, 7, 9]
```

NO SEPARATE THIS

- Before arrow functions, every new function defined its own this value based on how the function was called:
  - A new object in the case of a constructor.
  - *undefined* in strict mode function calls.
  - The base object if the function was called as an "object method”
- This proved to be less than ideal with an object-oriented style of programming.

```
function Person() {
        // The Person() constructor defines `this` as an instance of itself.
        this.age = 0;
        setInterval(function growUp() {
                // In non-strict mode, the growUp() function defines `this`
                // as the global object (because it's where growUp() is executed.),
                // which is different from the `this`
                // defined by the Person() constructor.
                this.age++;
        }, 1000);
}
var p = new Person();
```

- In ECMAScript 3/5, the *"this"* issue was fixable by assigning the value in this to a variable that could be closed over.

```
function Person() {
    var that = this; that.age = 0;
    setInterval(function growUp() {
        // The callback refers to the `that` variable of which
        // the value is the expected object.
        that.age++;
    }, 1000);
}
```

- Alternatively, a bound function could be created so that a preassigned *”this”* value would be passed to the bound target function (the growUp() function in the example above).
- An arrow function does not have its own this. The *”this”* value of the enclosing lexical scope is used; arrow functions follow the normal variable lookup rules. So while searching for *”this”* which is not present in the current scope, an arrow function ends up finding the *”this”* from its enclosing scope.
- Thus, in the following code, the *”this”* within the function that is passed to setInterval has the same value as the *”this”* in the lexically enclosing function:

```
function Person(){
        this.age = 0;
        setInterval(() => {
                this.age++; // |this| properly refers to the Person object
        }, 1000);
}
var p = new Person();
```

INVOKED THROUGH CALL OR APPLY

- Since arrow functions do not have their own *"this"*, the methods call() and apply() can only pass in parameters. Any *"this"* argument is ignored.

```
var adder = {
        base: 1,
        add: function(a) {
                var f = v => v + this.base;
                return f(a);
        },
        addThruCall: function(a) {
                var f = v => v + this.base;
                var b = { base: 2 };
                return f.call(b, a);
        }
};
console.log(adder.add(1)); // This would log 2
console.log(adder.addThruCall(1)); // This would log 2 still
```

NO BINDING OF ARGUMENTS

- Arrow functions do not have their own arguments object. Thus, in this example, *arguments* is simply a reference to the arguments of the enclosing scope:

```
var arguments = [1, 2, 3];
var arr = () => arguments[0];
arr(); // 1

function foo(n) {
        var f = () => arguments[0] + n; // foo's implicit arguments binding. arguments[0] is n
        return f();
}

foo(3); // 6
```

- In most cases, using rest parameters is a good alternative to using an *arguments* object.

```
function foo(n) {
        var f = (...args) => args[0] + n; return f(10);
}

foo(1); // 11
```

ARROW FUNCTIONS USED AS METHODS

- As stated previously, arrow function expressions are best suited for non-method functions. Let's see what happens when we try to use them as methods:

```
'use strict’;
var obj = {
         // does not create a new scope
        i: 10,
        b: () => console.log(this.i, this),
        c: function() {
        console.log(this.i, this);
        }
}
obj.b(); // prints undefined, Window {...} (or the global object)
obj.c(); // prints 10, Object {...}
```

- Arrow functions do not have their own this. Another example involving Object.defineProperty():

```
            'use strict’;
            var obj = {
                        a: 10
            };
            Object.defineProperty(obj, 'b', {
                        get: () => {
                        console.log(this.a, typeof this.a, this); // undefined 'undefined' Window {...} (or
                                                                    // the global object)
                        return this.a + 10; // represents global object 'Window', therefore 'this.a’
                                  // returns 'undefined’
                         }
            }
);
```

USE OF THE ”NEW” OPERATOR

- Arrow functions cannot be used as constructors and will throw an error when used with *new*.

```
var Foo = () => {};
var foo = new Foo();   // TypeError: Foo is not a constructor
```

USE OF "PROTOTYPE" PROPERTY

- Arrow functions do not have a *prototype* property.

```
var Foo = () => {};
console.log(Foo.prototype);   // undefined
```

USE OF THE "YIELD" KEYWORD

- The yield keyword may not be used in an arrow function's body (except when permitted within functions further nested within it). As a consequence, arrow functions cannot be used as generators.

FUNCTION BODY

- Arrow functions can have either a "concise body" or the usual "block body".
- In a concise body, only an expression is specified, which becomes the implicit return value. In a block body, you must use an explicit return statement.

```
var func = x => x * x;

 // concise body syntax, implied "return"

var func = (x, y) => { return x + y; };

// with block body, explicit "return" needed
```

RETURNING OBJECT LITERALS

- Keep in mind that returning object literals using the concise body syntax params => {object:literal} will not work as expected.

```
var func = () => { foo: 1 };
// Calling func() returns undefined!

var func = () => { foo: function() {} };
// SyntaxError: function statement requires a name
```

- This is because the code inside braces ({}) is parsed as a sequence of statements (i.e. foo is treated like a label, not a key in an object literal).
- You must wrap the object literal in parentheses:

```
var func = () => (
    { foo: 1 }
);
```

LINE BREAKS

- An arrow function cannot contain a line break between its parameters and its arrow.

```
var func = (a, b, c)
    => 1;
// SyntaxError: expected expression, got '=>"
```

- However, this can be amended by putting the line break after the arrow or using parentheses/braces as seen below to ensure that the code stays pretty and fluffy. You can also put line breaks between arguments.

```
var func = (a, b, c) => 1;
var func = (a, b, c) => ( 1 );
var func = (a, b, c) => { return 1 };
var func = ( a, b, c ) => 1; // no SyntaxError thrown
```

PARSING ORDER

- Although the arrow in an arrow function is not an operator, arrow functions have special parsing rules that interact differently with operator precedence compared to regular functions.

```
let callback;
callback = callback || function() {}; // ok

callback = callback || () => {};
// SyntaxError: invalid arrow-function arguments

callback = callback || (() => {}); // ok
```

REFERENCES

- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions