Arrows

Arrow functions (also called “fat arrow functions”) are undoubtedly one of the more popular features of ES6. They introduced a new way of writing functions.

Here is a function written in ES5 syntax:

```
function timesTwo(params) {
  return params * 2
}
timesTwo(4);  // 8
```

Now, here is the same function expressed as an arrow function:

```
var timesTwo = params => params * 2
timesTwo(4);  // 8
```

It’s much shorter! We can skip the curly braces and the return statement due to implicit returns (but only if there is no block — more on this below).

It is important to understand how the arrow function behaves differently compared to the regular ES5 functions.

One thing you will quickly notice is the change of syntaxes:

No parameters

If there are no parameters in the function, you can place empty parentheses before `=>`.

```
() => 42
```

You don’t even need the parentheses!

```
_ => 42
```

Single parameter

With these functions, parentheses are optional:

```
x => 42  || (x) => 42
```

Multiple parameters

Parentheses are required for these functions:

```
(x, y) => 42
```

Statements (as opposed to expressions)

With the arrow function, it is important to remember that statements in function need to have curly braces. Once the curly braces are present, you always need to write return as well to end the function

Here is an example of the arrow function used with an if statement:

```
var feedTheCat = (cat) => {
  if (cat === 'hungry') {
    return 'Feed the cat';
  } else {
    return 'Do not feed the cat';
  }
}
```

Block body

If your function is in a block of curly braces, you must also use the explicit return statement:

```
var addValues = (x, y) => {
  return x + y
}
```

Arrow Functions and this

In classic function expressions, the `this` keyword is bound to different values based on the `context` in which the function is called. Context is the object which calls the function. Whereas arrow functions use the value of this in their `lexical scope`. The concept behind lexical scope is that when a variable is bound to an environment, other functions that are defined in the same environment have access to that variable's value. This leads to very different behavior. For understanding the `context`, consider the following code:

```
var car = {
  make: "Lamborghini",
  model: "Huracán",
  fullName: function() {
    console.log(this.make + " " + this.model);
  }
};
car.fullName(); // Lamborghini Huracán
```

`car` is the object calling `fullname`. Thus `car` is the `fullname` context. So the value of `this` in fullname is bound to `car` (make:'Lamborghini' and model:'Huracan'). While working with the callback function, the traditional function creates unexpected issues. For example:

```
var car = {
    make:"Lamborghini",
    model:"Huracán",
    fullName:function () {
      console.log (this.make + " " + this.model);     // Lamborghini Huracán

      setTimeout(function(){
        console.log (this.make + " " + this.model)     // undefined undefined
      },1000);

    }
}
```

The **setTimeout** is a function that returns undefined as the context of setTimout function is the *window* object in the browser. Since there is no definition of `make` and `model` in the scope of the window object, the value of these 2 is taken to be undefined. Arrow functions don't bind the values of their variables with the context. Hence, if the same setTimeout function is called using the arrow function, it takes the value of `this` from its lexical scope. In the following example, the lexical scope of setTimeout is the function of `fullName`. For example:

```js var car = { make:"Lamborghini", model:"Huracán", fullName:function () { console.log (this.make + " " + this.model);

```
   setTimeout(()=>{
     console.log (this.make + " " + this.model)
   },1000);

}
```

} car.fullName(); // Lamborghini Huracán // Lamborghini Huracán ````

In the above code, the setTimeout function gets the value of `this` from the scope of function fullName. (make: Lamborghini, model: Huracan)

It is important to note that arrow functions are anonymous, which means that they are not named.

This anonymity creates some issues:

**Harder to debug**

When you get an error, you will not be able to trace the name of the function or the exact line number where it occurred.
```