Title: Arrow functions - Detailed Explanation - including this keyword

Answer:

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]
```

Description

Two factors influenced the introduction of arrow functions: shorter functions and non-binding of `this`.

Shorter functions

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

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

No separate `this`

Until arrow functions, every new function defined its own `this (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this)` value (a new object in the case of a constructor, undefined in strict mode (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode) function calls, the base object if the function is called as an "object method", etc.). 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 (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind) 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 execution context is used. Thus, in the following code, the `this` within the function that is passed to `setInterval` has the same value as `this` in the enclosing function:

```
function Person(){
  this.age = 0;

  setInterval(() => {
    this.age++; // |this| properly refers to the Person object
  }, 1000);
}

var p = new Person();
```

No binding of `arguments`

Arrow functions do not have their own `arguments` object (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/arguments). 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. argum
ents[0] is n
  return f();
}

foo(3); // 6
```

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 = {
  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` .

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
```

Tags: es6, functions / methods, javascript