DZone

# ES6 Arrow Functions In-Depth

by Can Ho MVB · Aug. 15, 16 · Web Dev Zone

Arrow functions are a new feature in ES6 for writing functions. This is one of the most favorite features in ES6 (see here). In this post, we're going to learn about this new feature.

## Syntax

Arrow functions, also known as **fat arrow** functions are functions that are defined with new syntax that use an arrow (=>):

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

const multiply = (x,y)=> {return x*y}
```

Parentheses are optional when there's only one parameter:

```
(param)=> { statements }//is equivalent to
param=> { statements }

const square = (x)=> {return x*x}
//is equivalent to
const square = x=> {return x*x}
```

However, when there are no parameters, parentheses are required:

```
()=> { statements }
```

When an expression (which produces value) is the body of an arrow function, braces are not needed:

```
(param1, param2, …, paramN)=> {return expression}
//is equivalent to
(param1, param2, …, paramN)=> expression


const multiply = (x,y)=> {return x*y}
//is equivalent to
const multiply = (x,y)=> x*y
```

**Note**: There must be no line break between the parameters and the arrow.

```
(param1, param2,..,paramN)
=> { statements } // Uncaught SyntaxError: Unexpected token =>
```

# Lexically Bound: this, super, new.target, and arguments

In normal functions, **this** is dynamically bound depending on how functions are called.

```
const Utils = {
  addAll: function(...) {
    ..
    this.add(..);
  },
  add: function(..) {
  }
}
...
Utils.addAll(...);//OK

let addAll= Utils.addAll;

addAll(...);// TypeError: this.updateUI is not a function(…)
```

Due to this behavior, it's very easy to lose track of **this** inside a function. In these cases, we need to workaround.

- **Work-around 1: self = this**

```
const Utils = {
  fetch: function(...) {
    var self = this;
    return $.ajax({
      ...
      success: function(){
        self.updateUI(..);
      }
    });
  },
  updateUI: function(..) {
  }
}
```

- **Work-around 2: Function.prototype.bind**

```
const Utils = {
```

```
  fetch: function(...) {
    return $.ajax({
      ...
      success: function(...){
        this.updateUI(..);
      }.bind(this)
    });
  },
  updateUI: function(..) {
  }
}
```

And the solution for this problem in ES6 is **arrow functions**.

```
const Utils = {
  fetch: function(...) {
    return $.ajax({
      ...
      success: ()=>this.updateUI(..)
    });
  },
  updateUI: function(..) {
  }
}
```

In addition to **this**, following objects are also bound lexically:

- **arguments**
- **super**
- **new.target**

# Anonymous Function Expressions

Arrow functions are of course functions:

```
const f = ()=>{}
typeof(f) //"function"
f instanceof Function // true
```

But they are function expressions. More than that, they don't have names. Thus, they are **anonymous function expressions**. And because of this, arrow functions shouldn't be used where we need function name for recursion and event binding.

# Can't Be Used as Constructors

Arrow functions can't be used as constructors. Using arrow functions as constructors will throw

Arrow functions can't be used as constructors. Using arrow functions as constructors will throw errors:

```
const f = ()=>{};
new f(); //TypeError: f is not a constructor
```

# No Prototype, Arguments, and Caller

As arrow functions can't be used as constructors, there is no need for **prototype** to be available. Arrow functions also don't have **arguments** and **caller**.

```
const f = ()=>{};
Object.getOwnPropertyNames(f); //["length", "name"]

const f0 = function(){}
Object.getOwnPropertyNames(f0); //["length", "name", "arguments", "caller", "prototype"]
```

# This Can't Be Changed

Arrow functions are "hard" bound. We can't change **this** value.

```
const f = ()=>console.log(this);
f();//window object
f.bind({});
f();//window object
f.call({});//window object
f.apply({});//window object
```

# Can't Be Used as Generators

In arrow functions, the keyword **yield** can't be used(except when normal functions are nested in it). Therefore, arrow functions can't be used as generators.

# When to Use/Not to Use

- Arrow functions should be used when we need a short function expression and that function expression doesn't rely on **this**. Using arrow functions with Array methods is a great example:

```
[1, 10, 2, 9, 3].sort((a,b)=>a-b); //[1, 2, 3, 9, 10]
```

- Arrow functions are best suited for non-method functions.

```
const Utils = {

```

```
  addAll: function(...) {
    ..
    this.add(..);
  },

  add: function(..) {
  },

  addFirst: (..)=> {
    ..
    this.add(..);
  }
}
..
Utils.addAll(); // OK

Utils.addFirst(); //  TypeError: this.add is not a function
```

The call **Util.addAll()** is fine because **addAll** is a normal function which is **this aware** (bound dynamically). Although we invoke as **Utils.addFirst()**, the reference to **this.add(..)** fails because **this** doesn't point to **Utils** as in normal functions. It inherits **this** from the surrounding scope. Thus, **arrow functions** should not be used as **method function**.

- Arrow functions can be used as nested functions which rely on **var self = this;** or **.bind(this)**
- In some cases, we have inner functions that reply on **arguments** object from the enclosing function, arrow functions are a best choice.

```
function list() {
  const args = Array.prototype.slice.call(arguments);
  const doSomething = function() {
    ...
    console.log(args);
  }
  doSomething();
}

// Better
function list() {
  const doSomething =()=>{
    ...
    console.log(arguments);
  }
  doSomething();
}
```

That is because inside **arrow functionsarguments** object inherits from the enclosing functions.

- Arrow functions shouldn't be used when we need function name reference for recursion and event binding.

# Summary

| | ARROW FUNCTIONS | NORMAL FUNCTIONS |
|---|---|---|
| **Binding** | Lexical | Dynamic |
| **this, arguments, super, new.target** | Don't have these objects. Inherit from enclosing scope | Have these objects. |
| **Type of function** | Anonymous function expression | Any type |
| **Used as constructor?** | ✗ | ✓ |
| **Own *prototype* object?** | ✗ | ✓ |
| **Can *this* be changed?** | ✗(hard bound) | ✓(soft bound) |
| **Used as generators?** | ✗ | ✓ |

**Note**: Apart from differences between arrow functions and normal functions mentioned above, all capabilities of normal functions are available to arrow functions, including default values, destructuring, rest parameters, etc..

```
[1, 2, 3, undefined].map((i=0)=>i*2)
// [2, 4, 6, 0]
```

Topics: ES6, SQUARE, EXPRESSION, FUNCTIONS, FEATURE, ARGUMENTS, ARROW FUNTIONS

Published at DZone with permission of Can Ho, DZone MVB. See the original article here.
Opinions expressed by DZone contributors are their own.