# ECMAScript 6

JS

# Scope & Context

Javascript

context and scope
are not the same

every function invocation has both a scope and a context associated with it

There is also a global scope
that resides on
top of every scope

scope is function-based

context is object-based

scope pertains to the variable access of a function when it is invoked

A scope is just a finite set of variables/objects that an execution context have access to

context is always the value of the ”this” keyword

which is a reference to the object that “owns” the currently executing code

The execution context is the *environment* of a *function* where it's code is executed

Every function has its own execution context

# Variable Scope

variable can be defined in either local or global scope

# What is “this” Context

context is most often determined by how a function is invoked

In JavaScript, “context”
refers to an object

Within an object, the keyword "**this**" refers to that object and provides an interface to the properties and methods that are members of that object

When a function is executed, the keyword “this” refers to the object that the function is executed in.

when a function is called as a method of an object, this is set to the object the method is called on

```
let obj = {
    foo: function() {
        return this;
    }
};

obj.foo() === obj; // true
```

when called as an unbound function, this will default to the global context or window object in the browser

```
function foo() {
    alert(this);
}

foo() // window or undefined is strict mode
```

```javascript
let drink = 'wine';

var foo = {
    drink: "beer",
    getDrink: function(){
        return drink;
    }
};

console.log( foo.getDrink() ); // wine
```

```javascript
let drink = 'wine';

let foo = {
    drink: "beer",
    getDrink: function(){
        return this.drink; // 'this' refers to the object "foo"
    }
};

console.log( foo.getDrink() ); // beer
```

JavaScript is a single threaded language

When the JavaScript interpreter initially executes code, it first enters into a global execution context

Each invocation of a function from this point on will result in the creation of a new execution context

For each execution context there is a scope chain coupled with it

```
function first() {
    second();
      function second() {
          third();
            function third() {
                fourth();
                  function fourth() {
                      // do something
                  }
            }
      }
}
first();
```

# Closures

```
function foo() {
    var localVariable = 'private variable';
    return function() {
        return localVariable;
    }
}

var getLocalVariable = foo();
getLocalVariable() // "private variable"
```

# Modules (pattern)

```
var Module = (function() {
    var privateProperty = 'foo';

    function privateMethod(args) {
        // do something
    }

    return {

        publicProperty: '',

        publicMethod: function(args) {
            // do something
        },

        privilegedMethod: function(args) {
            return privateMethod(args);
        }
    };
})();
```

# call(), apply() e bind()

```
function user(firstName, lastName, age) {
    // do something
}

user.call(window, 'John', 'Doe', 30);
user.apply(window, ['John', 'Doe', 30]);
```

# Exemplos

# Arrow functions

When (and why) you **should** use ES6 arrow functions—and when you **shouldn't**

Arrow functions are undoubtedly one of the more popular features of ES6.

```javascript
// ES5

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

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

# variations

```
// No Parameters
() => 42
//or
_ => 42

// single parameter
x => 42  || (x) => 42

// multiple parameteres
(x, y) => 42
```

In its most basic form, a *function expression* produces a value, while a *function statement (declaration)* performs an action.

With the arrow function, a statement need to have curly braces.

Once the curly braces are present, you always need to write return

# statement

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

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

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

If you are returning an object literal, it needs to be wrapped in parentheses

```
x =>({ y: x })
```

Main arrow function benefit:

No binding of ‘this’

In classic function expressions, the this keyword is bound to different values based on the *context* in which it is called.

```javascript
// ES5
var obj = {
  id: 42,
  counter: function counter() {
    setTimeout(function() {
      console.log(this.id);
    }.bind(this), 1000);
  }
};
```

With arrow functions however, this is *lexically bound*.

It means that it uses this from the code that contains the arrow function.

```javascript
// ES6
var obj = {
  id: 42,
  counter: function counter() {
    setTimeout(() => {
      console.log(this.id);
    }, 1000);
  }
};
```

ES6 arrow functions can't be bound to a this keyword, so it will lexically go up a scope, and use the value of this in the scope in which it was defined.

# When you should not use Arrow Functions

# Object methods

```
var cat = {
  lives: 9,
  jumps: () => {
    this.lives--;
  }
}
```

this is not bound to anything, and will inherit the value of this from its parent scope

# Callback functions with dynamic context

```
var button = document.getElementById('press');
button.addEventListener('click', () => {
  this.classList.toggle('on');
});
```

this is not bound to the button, but instead bound to its parent scope

# exemplo