Best Practices

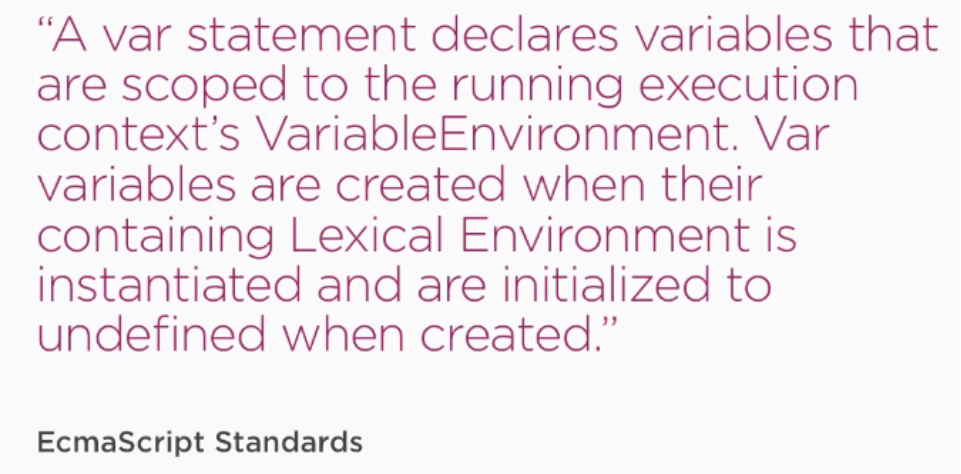
https://app.pluralsight.com/player?course=javascript-best-practices&author=jonathan-mills&name=javascript-best-practices-m0&clip=0&mode=live

Syntax:
Linting: A linter scans your code to detect potential problems and errors.
JS lint: first linting tool.
JS Hint is more configurable.
ESLint has a lot of configuration options.

== try to match things even if not the same time
=== strict doesn't try to match behind the scene.

To see safely if something exist or not:
typeof x , this will return undefined if doesn't exist yet.

Variables:
Hoisting: this is the JS default behaviour of moving all declarations to the top of the current scope.



All var declarations go to the top of your scope!
So global, at the top, in a funct scope, variable at the top of the function.
You don't need to initialise all the variables, just to declare them.

Functions:
A function is a first class object (like and integer, a string etc...) in JS. So you can assign them to a variable.
Hoisting is the same for functions. Declare them at the top too, even if you don't initialise them yet.

```
//variable first...
var x = 10;
//functions next
function print(input){
  //variables first
  var x = 0 //this goes here...
  //functions next
  function log(){
    //log stuff
  }
  //run code
  console.log(input);
}
//run code
print(10);
```

Behaviours:

Octals and Hexadecimals: if you write 012, JS will assume it's an octal
parseInt (number, base) to be sure.

With statement: avoid using them because scope is unsure (?)

to protect my scope:

```
'use strict';
var obj = {
  a: {
    b: {
      c: 'hello'
    }
  }
}

var c = 'this is important';

(function(newVar){
  console.log(newVar);
}(obj.a.b.c))
```

Async Patterns

Call backs
to avoid Xmas tree code with lots of anonymous function, it's better to give the function a name like: findAndValidateUser
This helps to understand better the code that is running.

Error handling: have a function with param err, and results:

```
function findAndValidateUser(username, password,
  done) {
  var url =
    'mongodb://localhost:27017/libraryApp';

  function validateUser(err, results) {
        if (results.password === password) {
          var user = results;
          done(null, user);
        } else {
          done(null, false, {
            message: 'Bad password'
          });
        }
```

done is a call back, when executing a call back, we want to be done, so use 'return' to make sure we're ending.

```
function findUser(err, db) {
  if (err) {
    return done(err, null)
  }
```

So when **doing callbacks**: 1. name functions, 2. use the pattern (err, results) 3. return your call backs

Promises

example of async methods, just to replicate an async (cb is for a call back)

```
function asyncMethod(message, cb){
  setTimeout(function(){
    console.log(message);
    cb();
  } , 500)
}
```

```
asyncMethod('Open DB Connection', function(){
  asyncMethod('Find User', function(){
    asyncMethod('validate User', function(){
      asyncMethod('do stuff', function(){})
    })
  })
})
```

this is not what we want, so will fix it with promises.

https://www.promisejs.org/

function asyncMethod (message, cb) {
	return new **Promise (function (fulfill, reject)** {});
	setTimeout ....(as above) but now instead of using cb at the end we will use fulfill
}

function asyncMethod (message, cb) {
	return new **Promise (function (fulfill, reject)** {
		setTimeout (function (){
			console.log(message);
			fulfill ();
		} , 500)
	});
}

A promise makes our function 'then-able"

```
function asyncMethod(message) {
  return new Promise(function (fulfill, reject) {
    setTimeout(function () {
      console.log(message);
      fulfill();
    }, 500)
  });
}

asyncMethod('Open DB Connection').then(function () {
  asyncMethod('Find User').then(function () {
    asyncMethod('validate User').then(function () {
      asyncMethod('do stuff').then(function () {})
    })
  })
})
```

to make it nicer (like for callbacks):

```
function findUser() {
  asyncMethod('Find User')
    .then(validateUser)
}

function validateUser() {
  asyncMethod('validate User')
    .then(doStuff)
}

function doStuff() {
  asyncMethod('do stuff')
    .then(function () {})
}
asyncMethod('Open DB Connection')
  .then(findUser)
```

or even better:

```
function findUser() {
  return asyncMethod('Find User')

}

function validateUser() {
  return asyncMethod('validate User')

}

function doStuff() {
  return asyncMethod('do stuff')
}
asyncMethod('Open DB Connection')
  .then(findUser)
  .then(validateUser)
  .then(doStuff)
  .then(function () {})
```

The reject, is like the err side of the code back.

Async Await

```
function asyncMethod(message) {
  return new Promise(function (fulfill, reject) {
    setTimeout(function () {
      console.log(message);
      fulfill();
    }, 500)
  });
}


async function main() {
  var one = await asyncMethod('Open DB Connection')
  var two = await asyncMethod('Find User')
  var three = await asyncMethod('validate User')
  var four = await asyncMethod('do stuff')

}

main();
```

to check compatibilities between ES5 and 6:
http://kangax.github.io/compat-table/es5/