ECMA6 Snippets

- Using ECMA6 we make optional properties explicit. By providing a default for a property, anyone reading the code sees the property is optional.

```
function complex({ name, email, phone = null }) {
  //variables `name`, `email`, and `phone` are available here
}

var user = {
  name: 'foo',
  email: 'bar',
  phone: '123456'
};
complex(user);
```

- Using Object initialiser short hand you can avoid duplication.This feature allows you to skip repeating yourself in object literals.

  A pattern seen in the nodejs world:

```
module.exports = {
  someFunction: someFunction,
  otherFunction: otherFunction
};
```

  Sometimes you also see the same with the module pattern:

```
var myModule = (function() {
  /* declarations here */

  return {
    someFunction: someFunction,
    otherFunction: otherFunction
  };
})();
```

  Using ES6's object initializer shorthand, we can rewrite it:

```
module.exports = {
  someFunction,
  otherFunction
};
```

```
var myModule = (function() {
  /* declarations here */

  return {
    someFunction,
    otherFunction
  };
})();
```

- Promises allow easily composable asynchronous functions, without creating a callback-pyramids like in the first example. We do need a bit more code to create the promise in the first place, but if our asynchronous functions were longer, it wouldn't be such a visible difference.

Let’s look at a simple example to show why promises can be useful vs. callbacks. Say you have three asynchronous functions and you need to run all three, and then execute another function:

```
//for sake of example, we're just using setTimeout. This could be any asynchronous action as
well.
function asyncA(callback) { setTimeout(callback, 100); }
function asyncB(callback) { setTimeout(callback, 200); }
function asyncC(callback) { setTimeout(callback, 300); }

asyncA(function(resultA) {
  asyncB(function(resultB) {
    asyncC(function(resultC) {
      someFunction(resultA, resultB, resultC);
    });
  });
});
```

Compare to an implementation using promises:

```
function promiseA() { return new Promise((resolve, reject) => setTimeout(resolve, 100)); }
function promiseB() { return new Promise((resolve, reject) => setTimeout(resolve, 200)); }
function promiseC() { return new Promise((resolve, reject) => setTimeout(resolve, 300)); }

Promise.all([promiseA(), promiseB(), promiseC()]).then(([a, b, c]) => {
  someFunction(a, b, c);
});
```

```
function timeout(duration = 0) {
    return new Promise((resolve, reject) => {
        setTimeout(resolve, duration);
    })
}

var p = timeout(1000).then(() => {
    return timeout(2000);
}).then(() => {
    throw new Error("hmm");
}).catch(err => {
    return Promise.all([timeout(100), timeout(200)]);
})
```

- Arrow functions
  The syntax is quite flexible:

```
//arrow function with no parameters
var a1 = () => 1;

//arrow with one parameter can be defined without parentheses
var a2 = x => 1;
var a3 = (x) => 1;

//arrow with multiple params requires parentheses
var a4 = (x, y) => 1;

//arrow with body has no implicit return
var a5 = x => { return 1; };
```

- *this* binding

  How often do you see something like this done in JS?

```
var self = this;
el.onclick = function() {
  self.doSomething();
};
```

  This pattern crops up occasionally, especially in less experienced dev's code. The solution is to use `bind` to fix `this` in the function, but it's a bit verbose.

  With arrows, we can instead just do this:

```
el.onclick = () => this.doSomething()
```

- Working with iterators

  ES6 also provides a new looping syntax called for-of:

```
var arr = [1, 2, 3];
for(var value of arr) {
  console.log(value);
}
//output: 1, 2, 3
```

- Implicit return
  A statement body allows multiple statements.

```
el.onclick = (x, y, z) => {
  foo();
  bar();
  return 'baz';
}

//equivalent to:
el.onclick = function(x, y, z) {
  foo();
  bar();
  return 'baz';
}.bind(this);
```

- An expression body allows only a single expression, and has an implicit return. This means it's great for functional-style code, for example when using `Array#map` or`Array#filter`

```
//ES5 style
var names = users.map(function(u) { return u.name; });

//ES6 style
var names = users.map(u => u.name);
```

- Tail calls: Calls in tail-position are guaranteed to not grow the stack unboundedly. Makes recursive algorithms safe in the face of unbounded inputs.

```
function factorial(n, acc = 1) {
    'use strict';
    if (n <= 1) return acc;
    return factorial(n - 1, n * acc);
}

// Stack overflow in most implementations today,
// but safe on arbitrary inputs in ES6
factorial(100000)
```

- Math + Number + String + Array + Object APIs
  Many new library additions, including core Math libraries, Array conversion helpers, String helpers, and Object.assign for copying.

```
Number.EPSILON
Number.isInteger(Infinity) // false
Number.isNaN("NaN") // false

Math.acosh(3) // 1.762747174039086
Math.hypot(3, 4) // 5
Math.imul(Math.pow(2, 32) - 1, Math.pow(2, 32) - 2) // 2

"abcde".includes("cd") // true
"abc".repeat(3) // "abcabcabc"

Array.from(document.querySelectorAll('*')) // Returns a real Array
Array.of(1, 2, 3) // Similar to new Array(...), but without special one-arg behavior
[0, 0, 0].fill(7, 1) // [0,7,7]
[1, 2, 3].find(x => x == 3) // 3
[1, 2, 3].findIndex(x => x == 2) // 1
[1, 2, 3, 4, 5].copyWithin(3, 0) // [1, 2, 3, 1, 2]
["a", "b", "c"].entries() // iterator [0, "a"], [1,"b"], [2,"c"]
["a", "b", "c"].keys() // iterator 0, 1, 2
["a", "b", "c"].values() // iterator "a", "b", "c"

Object.assign(Point, { origin: new Point(0,0) })
```