ES6

Classes

Classes are a new feature in ES6, used to describe the blueprint of an object and make EcmaScript's prototypical inheritance model function more like a traditional class-based language.

```
class Hamburger {
  constructor() {
    // This is the constructor.
  }
  listToppings() {
    // This is a method.
  }
}
```

Object

An object is an instance of a class which is created using the `new` operator. When using a dot notation to access a method on the object, `this` will refer to the object to the left of the dot.

```
let burger = new Hamburger();
burger.listToppings();
```

A Refresher on `this`

Inside a JavaScript class we'll be using `this` keyword to refer to the instance of the class. E.g., consider this case:

```
class Toppings {
  ...

  formatToppings() { /* implementation details */ }

  list() {
    return this.formatToppings(this.toppings);
  }
}
```

However, `this` can also refer to other things. There are two basic cases that you should remember.

1. Method invocation:

```
someObject.someMethod();
```

Here, `this` used inside `someMethod` will refer to `someObject`, which is usually what you want.

2. Function invocation:

```
someFunction();
```

Arrow Functions

ES6 offers some new syntax for dealing with `this` : "arrow functions".
Arrow functions also make higher order functions much easier to work with.

The new "fat arrow" notation can be used to define anonymous functions in a simpler way.

Consider the following example:

```
items.forEach(function(x) {
  console.log(x);
  incrementedItems.push(x+1);
});
```

This can be rewritten as an "arrow function" using the following syntax:

```
items.forEach((x) => {
  console.log(x);
  incrementedItems.push(x+1);
});
```

Template Strings

In traditional JavaScript, text that is enclosed within matching `"` or `'` marks is considered a string. Text within double or single quotes can only be on one line. There was no way to insert data into these strings. This resulted in a lot of ugly concatenation code that looked like:

```
var name = 'Sam';
var age = 42;

console.log('hello my name is ' + name + ' I am ' + age + ' years old');
```

ES6 introduces a new type of string literal that is marked with back ticks (`). These string literals *can* include newlines, and there is a string interpolation for inserting variables into strings:

```
var name = 'Sam';
var age = 42;

console.log(`hello my name is ${name}, and I am ${age} years old`);
```

Spread Syntax

Spread example:

```
const add = (a, b) => a + b;
let args = [3, 5];
add(...args); // same as `add(args[0], args[1])`, or `add.apply(null, args)`
```

Functions aren't the only place in JavaScript that makes use of comma separated lists - arrays can now be concatenated with ease:

```
let cde = ['c', 'd', 'e'];
let scale = ['a', 'b', ...cde, 'f', 'g'];  // ['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

Similarly, object literals can do the same thing:

```
let mapABC  = { a: 5, b: 6, c: 3};
let mapABCD = { ...mapABC, d: 7};  // { a: 5, b: 6, c: 3, d: 7 }
```

Destructuring

Destructuring is a way to quickly extract data out of an `{}` or `[]` without having to write much code.

To borrow from the MDN, destructuring can be used to turn the following:

```
let foo = ['one', 'two', 'three'];

let one   = foo[0];
let two   = foo[1];
let three = foo[2];
```

into

```
let foo = ['one', 'two', 'three'];
let [one, two, three] = foo;
console.log(one); // 'one'
```