**ES6 Reminders and a Few New Pieces of Syntax**

**References – 3 short and excellent sites that compare classic JS with ES6 and briefly detail the new features.**  (You saw all these in Chapter 5.) This is just in case you have questions about syntax, or want examples of some of these features.

* <https://adrianmejia.com/overview-of-javascript-es6-features-a-k-a-ecmascript-6-and-es2015/>
* <http://es6-features.org/#Constants>
* <https://www.freecodecamp.org/news/these-are-the-features-in-es6-that-you-should-know-1411194c71cb/>

The reason for this document is so that you can recognize some syntactic structures which were introduced in ES6. You will not be writing code (unless you choose to) with these "new to you" features, but you are likely to read code with them. So you should recgonize them --- and they're not hard.

**Reminders**

* **Block scope variables**
  + By now you are very familiar with the use of **let** and **const.**
* **Arrow function**
  + Please review the Arrow Function Syntax in Chapter 5   
    <http://web.simmons.edu/~menzin/CS321/Unit_2_JavaScript_and_HTML_Forms/Chapter_5_HTML5_Innovations/>
  + Remember that arrow functions are function expressions, so they are not hoisted and they are not appropriate for constructors.
  + Please also remember that in an arrow function *this* has lexical scope – that is, the value of *this* depends on where it is written in the syntax and not on how the function is called.
  + Arrow function syntax occurs all over the place in node.js
* **Template literals** 
  + The page on this is also in Chapter 5.
  + You can use template literals to construct multi-line strings – see <https://adrianmejia.com/overview-of-javascript-es6-features-a-k-a-ecmascript-6-and-es2015/#Multi-line-strings> for examples.
* **Default parameters**
  + Like template literals, these are "syntactic sugar" and they are very easy to use.
  + If you need a reminder, go to any of the 3 references at the top of this page.

**New to you features  
 *Modules*** are new and important.   
  
***Promise****s* (for asynchronous code) are improved in ES6. We haven't talked about them yet, but we should have a little more experience with Node.js before we do so.  
  
The ***other features*** are about making your coding more efficient, rather than introducing new functionality.   
As with arrow functions, their appearance is odd at first glance (sometimes at second glance too.)   
  
I want to emphasize that ***you need to be able to read code that others have written with these features, but writing code with rest and spread parameters or destructuring is optional.***As far as I am concerned, classes are of interest mainly to Java programmers who never managed to learn how inheritance works in JavaScript. Feh!

1. **New array methods** – a few easy and new to you methods. These are all similar and easy to use: filter(), find(), every() etc.   
   Read about filter() at <https://www.javascripttutorial.net/javascript-array-filter/> to see how you can take put only the elements you are interested in from some array *into another array.*Please notice the syntax:  
   let newArray = oldArray.filter(callback, context);  
   The context is optional (& sets the value of *this.)*The callback is often written as an arrow function:  
    element => *boolean condition to test for inclusion*Go thru the linked example slowly b/c the syntax is very compact.  
   Then go to <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/every> and look at the other functions in the menu on the left. Be sure to examine the syntax and examples for at least *find(), forEach(), map() and reduce().* These, and *filter*(), are all used often, and they are quite straightforward.  
   Of course, you are already familiar with sort() and some of the other methods.
2. **Destructuring**
   * Destructuring is a way to extract values from an object and assign them to variables with the same name.  
     JD Issaaks' book "Get Programming with JavaScript Next" gives the following example:

let person = {  
 name: 'Christina',  
 age: 25  
}

let { name, age } = person;

console.log(name, age);

**Do:** Type this code into your browser console and you should get   
 Christina 25.  
  
The critical line here is *let {name, age} = person;*

What ES6 has done is declare two new let variables, and assign to them the values associated with those names in the person object.

In other words *let {name, age} = person;* is equivalent to:  
 let name = person.name;  
 let age = person.age;

* + We can do fancier things with nested objects --- **read sections 11.1 and 11.2 on destructuring objects and arrays in** [**https://learning.oreilly.com/library/view/get-programming-with/9781617294204/kindle\_split\_021.html**](https://learning.oreilly.com/library/view/get-programming-with/9781617294204/kindle_split_021.html)and browse 11.3 and 11.4 to see how fancy it can get.  
    If you want even more complicated examples, go to <https://leanpub.com/understandinges6/read/#leanpub-auto-destructuring-for-easier-data-access>

1. **Rest and spread parameters**The **spread operator** takes an iterable object (e.g. an array) and spreads it out into individual values.  
   The **rest parameters** allow you to do the opposite – ie. gather then together.  
     
   The **spread** operator, like the rest parameters, uses 3 dots **…**
2. The **spread** operator is a replacement for the classic *arguments* parameter (which put all the parameters into an array, so you could operate on them – especially useful when you had a variable number of parameters.)  
     
   The formal description is at [Spread syntax (...) - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax) and some excellent examples are at [JavaScript ES6 - Spread Operator And Rest Parameter (c-sharpcorner.com)](https://www.c-sharpcorner.com/blogs/javascript-es6-spread-operator-and-rest-parameter) and [The Practical Usages of JavaScript Spread Operator (javascripttutorial.net)](https://www.javascripttutorial.net/es6/javascript-spread/) You need to be able to recognize what such code does --- writing it is optional.  
     
     
   The formal description of **rest parameters** is at [Rest parameters - JavaScript | MDN (mozilla.org)](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters) (you don't need to worry about how it differs from arguments). But you will get a better idea of its use at [JavaScript ES6 - Spread Operator And Rest Parameter (c-sharpcorner.com)](https://www.c-sharpcorner.com/blogs/javascript-es6-spread-operator-and-rest-parameter) - the examples for the **rest** parameters are below those for the spread operator..  
     
   There is also a section on "Spread/Rest" near the start of Chapter 2 of Kyle Simpson's ES6 and Beyond book <https://learning.oreilly.com/library/view/you-dont-know/9781491905241/ch02.html>   
     
   ***What you should do****:* **Read** the very short section in the Kyle Simpson book (you can stop when he gets to discussing the deprecated *arguments* parameter) and look at all the examples in [JavaScript ES6 - Spread Operator And Rest Parameter (c-sharpcorner.com)](https://www.c-sharpcorner.com/blogs/javascript-es6-spread-operator-and-rest-parameter)  
   **Explain to yourself** the examples in this link and bring any questions to class!  
     
      
   Again, I want to emphasize that you need to be able to recognize the code, but writing it is optional.
3. **Modules**Modules are important both as a way to organize large, complex code bases and also for Node.js So what are they?

Modules are a way to put together a chunk of code which has the following characteristics:

* + You specify with the **export** statement exactly what values and function will be available to users of the module (i.e. those who **import** from the module.)
  + All the rest of the code in the module is *private.*For example, a module might export a sort method for arrays, but inside the module would be several helper functions (which are not made available outside the module) that sort different size arrays using different methods
  + The identifiers in the module do not become part of the global namespace.
  + You may choose to export only one value or function (in which case it will have the same name as the module, and it is a *default* export), or multiple values and functions (in which case they will be packaged together in an object, with their names as the keys, and their values … as the values.)
  + The code is automatically in *strict*  mode. We haven't discussed strict mode, but you can read about it at <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode>

Again,, while we won't be writing modules ourselves, you should be familiar with the basic ideas and recognize the syntax. Fortunately, there is a short and clear description in Chapter 13 of Understanding ECMAScript6 at <https://learning.oreilly.com/library/view/understanding-ecmascript-6/9781492017509/xhtml/ch13.xhtml> .   
So please **read** that chapter at least up to "Re-exporting a binding".  
  
Note: When we come to Node very, very soon, you will see that the **require()** method is used to import a module. This is because Node.js uses the syntax of the module system developed in CommonJS. (This is a little like ES6 incorporating some of the structures which jQuery invented.)  
  
We'll get to look at a number of modules in detail as we dive into Node.js

1. **Classes (more syntactic sugar- it looks different but inheritance is still from prototypes)**Read Chapter 9 of <https://learning.oreilly.com/library/view/understanding-ecmascript-6/9781492017509/xhtml/ch09.xhtml> up to the section on "Why Use Class Syntax?". The rest of the chapter points out that class definitions are not hoisted, and that they may be written as either functions or function expressions. We will not be reading or writing classes in this course.   
   I think there is (or at least initially was) a bit of disdain for programmers who used classes – sort of "real JS programmers don't use classes" (because you still have prototypical inheritance.) That may change over time.
2. **Promises**

As noted above, we will defer this topic until we have done a little with Node.js I just note here that when we wrote ajax() requests with jQuery we had to consider the fact that JavaScript is asynchronous, and that sometimes we wanted to defer using some code we had retrieved until all of the callback had finished executing. In jQuery's ajax() calls we were able to achieve that with always() method. Promises provide that kind of handling of aynchronicity issues in a more general context.