ES6 Syntax Udacity Full

1. Let & Const
2. Template Literals
3. Desctructuring
4. Object Literal
5. Looping (for ... of loop)
6. Spread... Operator
7. ...Rest parameter
8. Arrow Function () = > {};
9. This. Regular Functions()
10. This. Arrow Functions()
11. Classes
12. ES6 Syntax Deploy

Let & Const

If a variable is declared using `let` or `const` inside a block of code (denoted by curly braces { }), then the variable is only valid between the curly braces. Outside it is undefined variable due to the block scoping.

- use let when you plan to reassign new values to a variable, and
- use const when you don't plan on reassigning new values to a variable.

`Const` note:

- arrays are reference type objects so if we are adding a new item with array.push, we can modify it.
- ojbect are reference type, we are changing the value where the reference is pointed.

Template Literals

Template literals are essentially string literals that include embedded expressions.
Using `(``)` & `${variable}`

```
//Before
let message = student.name + ' please see ' + teacher.name + ' in ' + teacher.room + ' to pick
 up your report card.';
//After
let message = `${student.name} please see ${teacher.name} in ${teacher.room} to pick up your re
port card.`;
```

Template Literals preserve newlines as part of the string!

```
//Old way
```

```
let note = teacher.name + ',\n\n' +
  'Please excuse ' + student.name + '.\n' +
  'He is recovering from the flu.\n\n' +
  'Thank you,\n' +
  student.guardian;


//New way
let note = '${teacher.name},
  Please excuse ${student.name}.
  He is recovering from the flu.
  Thank you,
  ${student.guardian}`;
```

Desctructuring

Array:
You don't have to specify any indexes to extract values from the array

```
const point = [10, 25, -34];
const [x, y, z] = point;
console.log(x, y, z);                //-->10,25,-34
// You can also ignore values when destructuring arrays.
 const [a, , b] = point;
 console.log(a,b);                    //10,-34
```

Object:

```
const gemstone = {
  type: 'quartz',
  color: 'rose',
  karat: 21.29
};
//Object propertynames has to match the variable names
```

```
const {type, color, karat} = gemstone;

console.log(type, color, karat);
```

Note:

1. After destructuring `this.` keyword will not referencing the same object, thus the function will not work properly
2. Important to notice that the variable names should corresponding with the object property names
3. Important that the object property are referenced by names, not by position!
4. You can rename the desctructed functions.

```
const circle = {

  radius: 10,

  color: 'orange',

  getArea: function() { return Math.PI * this.radius * this.radius;  },

  getCircumference: function() {  return 2 * Math.PI * this.radius;  }

};

//*1* let {radius, getArea, getCircumference} = circle;

//*2* let {radius, getArea1} = circle;     //-->Fail

//*3* let {radius, ,getArea} = obj;      --->Fail

//*4* let {radius, getArea: area}
```

Object Literal

Properties: (Easy object generation)

```
let type = 'quartz';

let color = 'rose';

let carat = 21.29;

const gemstone = {

  type: type,

  color: color,

  carat: carat

};

console.log(gemstone);


//ES6 if the properties have the same name as the variables being assigned to them.
```

```
const gemstone = {type,color,carat};
```

(+ dynamic keys): (Object's propertyName as variable)

```
let type = 'quartz';
let color = 'rose';
let gemCarat = 'carat'
const gemstone = {
  type: type,
  color: color,
  [gemCarat]:21.29
};
```

Functions:

```
const gemstone = {
  carat,
  calculateWorth: function() {...}
};
let gemstone = {
  carat,
  calculateWorth() { ... },
  "calculate worth"() {...}                //Functions can be named with spaces in it's name!
};
console.log(gemstone.calculateworth);
console.log(gemstone.["calculate worth"]);
```

Looping (for ... of loop)

- You can loop through any iterable data
- You can stop or break a for...of loop at anytime

```
const digits = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

//You can loop through any iterable data

for (const digit of digits) {

  if (digit % 2 === 0) {

    continue;

  }

  console.log(digit);

}

/*Prints:

1

3

5

7

9*/
```

Spread... Operator

Gives you the ability to expand, or spread, iterable objects into multiple elements.

```
const fruits = ["apples", "bananas", "pears"];

const vegetables = ["corn", "potatoes", "carrots"];


//Merge the two arrays

const produce = [fruits...,vegetables...];

const produce = fruits.concat(vegetables);
```

...Rest parameter

Represent an indefinite number of elements as an array

Desctructuring arrays

```
const order = [20.17, 18.67, 1.50, "cheese", "eggs", "milk", "bread"];

const [total, subtotal, tax, ...items] = order;
```

```
console.log(total, subtotal, tax, items);

//Prints: 20.17 18.67 1.5 ["cheese", "eggs", "milk", "bread"]
```

Variadic functions are functions that take an **indefinite number of arguments**.

```
function sum(...nums) {
  let total = 0;
  for(const num of nums) {
    total += num;
  }
  return total;
}
```

Functions

Arrow Function () = > {};

```
const upperizedNames = ['Farrin', 'Kagure', 'Asser'].map(function(name) {
  return name.toUpperCase();
});
const upperizedNames = ['Farrin', 'Kagure', 'Asser'].map(
  name => name.toUpperCase()
);
```

Multiple parameters requires parentheses

```
const orderIceCream = (flavor, cone) => console.log(`Here's your ${flavor} ice cream in a ${con
e} cone.`);
orderIceCream('chocolate', 'waffle');
//Prints: Here's your chocolate ice cream in a waffle cone.
```

Concise and block body syntax

```
//Concise body syntax    (does not include return)
const upperizedNames = ['Farrin', 'Kagure', 'Asser'].map(
  name => name.toUpperCase()
);
//Block body syntax     (curly braces +  return statement needs to be used to actually return s
omething from the function.)
const upperizedNames = ['Farrin', 'Kagure', 'Asser'].map( name => {
  name = name.toUpperCase();
  return `${name} has ${name.length} characters in their name`;
});
```

This. Regular Functions()

The value of the this keyword is based completely on how its function (or method) is called. this could be any of the following:

1. A new object (If the function is called with new)

```
const mySundae = new Sundae('Chocolate', ['Sprinkles', 'Hot Fudge']);
```

In the code above, the value of this inside the **Sundae** constructor function is a new object because it was called with new.

2. A specified object (If the function is invoked with call/apply)

```
const result = obj1.printName.call(obj2);
```

In the code above, the value of this inside printName() will **refer to obj2** since the first parameter of call() is to explicitly set what this refers to.

3. A context object (If the function is a method of an object)

```
data.teleport();
```

In the code above, the value of this inside teleport() will refer to data.

4. The global object or undefined (If the function is called with no context)

```
teleport();
```

In the code above, the value of this inside teleport() is either the global object or, if in strict mode, it's undefined.

`This.` Arrow Functions()

With arrow functions, the value of this is based on the **function's surrounding context**. In other words, the value of this inside an arrow function is the same as the value of this outside the function.

Regular function example (Incorrect referencing)

```
//The function passed to setTimeout() is called without new, without call(), without apply(), a
nd without a context object. That means the value of this inside the function is the global obj
ect and NOT the dessert object.
// constructor
function IceCream() {
  this.scoops = 0;
}
// adds scoop to ice cream
IceCream.prototype.addScoop = function() {
  setTimeout(function() {
    this.scoops++;
    console.log('scoop added!');
  }, 500);
};
const dessert = new IceCream();
dessert.addScoop();       // Prints "scoop added", but scoops will remain 0 since the this. key
word is referening the window
```

Regular function example (Using closure)

```
//It sets the cone variable to this. and then looks up the cone variable when the function is
 called.
function IceCream() {
  this.scoops = 0;
}
```

```
// adds scoop to ice cream
IceCream.prototype.addScoop = function() {
  const cone = this; // sets `this` to the `cone` variable
  setTimeout(function() {
    cone.scoops++; // references the `cone` variable
    console.log('scoop added!');
  }, 0.5);
};
const dessert = new IceCream();
dessert.addScoop();         // Prints "scoop added" and scoops will change to 1 since icecream
 is referenced
```

Well that's exactly what arrow functions do. It is searching for the value of `this`. in the function's surrounding context.

```
//  An arrow function is passed to setTimeout, wich will check the surronding context, here ins
ide the iceCream.prototype.addScoop this. refers to the iceCream function.
function IceCream() {
  this.scoops = 0;
}
// adds scoop to ice cream
IceCream.prototype.addScoop = function() {
  setTimeout(() => { // an arrow function is passed to setTimeout, wi
    this.scoops++;
    console.log('scoop added!');
  }, 0.5);
};
const dessert = new IceCream();
dessert.addScoop();
```

Arrow function example (Global context)

```
// Arrow functions inherit their this value from their surrounding context. Outside of the addS
coop() method, the value of this is the global object.
function IceCream() {
    this.scoops = 0;
}
// adds scoop to ice cream
IceCream.prototype.addScoop = () => { // addScoop is now an arrow function
  setTimeout(() => {
    this.scoops++;
    console.log('scoop added!');
  }, 0.5);
};
const dessert = new IceCream();
dessert.addScoop();
```

Default Function Parameters:

```
function greet(name = 'Student', greeting = 'Welcome') {
  return `${greeting} ${name}!`;
}
greet(); // Welcome Student!
greet('James'); // Welcome James!
greet('Richard', 'Howdy'); // Howdy Richard!
```

Set default array parameters:

```
function createGrid([width = 5, height = 5] = []) {
  return `Generating a grid of ${width} by ${height}`;
}
createGrid(); // Generates a 5 x 5 grid
```

Set default object parameters:
(Thus you can reference keywords of the parameters, instead of the positions, while using array parameters)

```
//Object default parameters, can be referenced with keywords

function createSundae({scoops = 1, toppings = ['Hot Fudge']} = {}) { … }

createSundae({toppings: ['Hot Fudge', 'Sprinkles', 'Caramel']});

//Array default parameters has to be referenced with positions.

function createSundae([scoops = 1, toppings = ['Hot Fudge']] = []) { … }

createSundae([undefined, ['Hot Fudge', 'Sprinkles', 'Caramel']]);
```

Classes

ES5 Classes with prototype inheritance

```
// Constructor, starts with Big first capital letter

function Plane(numEngines) {

  this.numEngines = numEngines;

  this.enginesActive = false;

}

// methods "inherited" by all instances, adding a startEngines method to all of the instances o

f the inherited functions

Plane.prototype.startEngines = function () {

  console.log('starting engines...');

  this.enginesActive = true;

};

// Constructor hast o be initated with the new keyword

const richardsPlane = new Plane(1);

richardsPlane.startEngines();

const jamesPlane = new Plane(4);

jamesPlane.startEngines();
```

Things to note:

- The constructor function is called with the new keyword
- The constructor function, by convention, starts with a capital letter
- The constructor function controls the setting of data on the objects that will be created
- "Inherited" methods are placed on the constructor function's prototype object.

ES6 Syntax:

```
//Basically we are creating a class with 1 constructor function + 1 named function
class Plane {
//Everytime an instance (objects) of this class is created the constructor function will be cal
led, and the passed parameters will be used.
  constructor(numEngines) {
    this.numEngines = numEngines;
    this.enginesActive = false;
  }
  startEngines() {
    console.log('starting engines…');
    this.enginesActive = true;
  }
}
typeof Plane; // function     --< it is still a function!
```

Things to note:

- Plane is still a function!
- There are no comma separators like in the objects.
- The constructor function controls the setting of data on the objects that will be created
- "Inherited" methods are placed on the constructor function's prototype object.

Static Method:

```
class Plane {
  constructor(numEngines) {
    this.numEngines = numEngines;
    this.enginesActive = false;
  }
  static badWeather(planes) {h
    for (plane of planes) {
      plane.enginesActive = false;
    }
  }
  startEngines() {
```

```
    console.log('starting engines…');
    this.enginesActive = true;
  }
}
const richardsPlane = new Plane(1);                    //Create an instance of the class
richardsPlane.startEngines();                          //Calling on of the declared functio
ns
Plane.badWeather();                                    //Can be called directly on the clas
s (since it is connected with the prototype)
```

Subclasses with ES6:
Notes:

1. By using `extend` you can create a subclass from a simple class
2. By using `super`, you can inherit **property assignement** (this.size = size), functions from the parent class
3. `Super` must be called before `this`

```
class Tree {
  constructor(size = '10', leaves = {spring: 'green', summer: 'green', fall: 'orange', winter:
null}) {
    this.size = size;
    this.leaves = leaves;
    this.leafColor = null;
  }
  changeSeason(season) {
    this.leafColor = this.leaves[season];
    if (season === 'spring') {
      this.size += 1;
    }
  }
}
class Maple extends Tree {                             // 1. Creates a subclass from
 the parentclass of Tree.
  constructor(syrupQty = 15, size, leaves) {
      //****    3     *
```

```
    super(size, leaves);                                    // 2.1 inherit the size, leav
es properties
    this.syrupQty = syrupQty;
  }
  changeSeason(season) {
    super.changeSeason(season);                             //2.2 Inherit the changeSeaso
n function of the Tree class
    if (season === 'spring') {
      this.syrupQty += 1;
    }
  }
  gatherSyrup() {
    this.syrupQty -= 3;
  }
}
const myMaple = new Maple(15, 5);
myMaple.changeSeason('fall');
myMaple.gatherSyrup();
myMaple.changeSeason('spring');
```

ES6 Syntax Deploy

What is a polyfill?
A polyfill, or polyfiller, is a piece of code (or plugin) that provides the technology that you, the developer, expect the browser to provide natively.

Sample

```
if (!String.prototype.startsWith) {
  String.prototype.startsWith = function (searchString, position) {
    position = position || 0;
    return this.substr(position, searchString.length) === searchString;
  };
```

```
}
```

As you can see, a polyfill is just regular JavaScript.

Link for al polyfills:
https://github.com/Modernizr/Modernizr/wiki/HTML5-Cross-Browser-Polyfills

Complier (Source code ---> Machine Code)
Transpiler (Source code ES6---> Source code ES5)

Tutorial for Babel package (Node Js)
https://www.codementor.io/iykyvic/writing-your-nodejs-apps-using-es6-6dh0edw2o