**ES6**

Harmony === ES6 === ES2015

Q: debugger statement?

A: debugger;

**LET, CONST & VAR**

Hoisting:

\_Before JS code is executed, all variable (var) is hoisted to the top of the function scope (globally or locally to an entire functions scope)

Q: but for this piece of code

**function** **getClothing**(isCold) {

**if** (isCold) {

**var** freezing = 'Grab a jacket!';

} **else** {

**var** hot = 'It’s a shorts kind of day.';

console.log(freezing);

}

}

when ‘freezing’ and ‘hot’ are hoisted, do they carry their value (‘grab a jacket’) and ‘it’s shorts kind of day’ with them?

A: No, it’s only declared but not given a value at the beginning

Q: Why was that function’s console.log is undefined?

A: Because since it’s **false**, freezing is never assigned a value. Only when it’s true (or isCold ===true), then the value of freezing would be defined

Q: Let/ const vs Var

So if I use ‘let’ or ‘const inside a block of code (inside curly braces), the variables are trapped within the block they’re declared

Otherwise variable with var is hoisted up (declared only but not with the value)

DO NOT use var at all

Q: Let vs Const?

A: +)Variables declared with “let” can be reassigned but can’t be redeclared in the same scope

+) With Const, variables must be assigned an initial avalue but cannot be redeclared or reassigned

* Const is the strictest way to declare a variable

Ex:

Let friend = ‘Sarah’;

friend = ‘Maya’;

console.log(friend)

* Will be Maya

Q: So in this test, when to use “let”, and when to use “const”?

var CHARACTER\_LIMIT = 255;

var posts = [

"#DeepLearning transforms everything from self-driving cars to language translations. AND it's our new Nanodegree!",

"Within your first week of the VR Developer Nanodegree Program, you'll make your own virtual reality app",

"I just finished @udacity's Front-End Web Developer Nanodegree. Check it out!"

];

// prints posts to the console

function displayPosts() {

for (var i = 0; i < posts.length; i++) {

console.log(posts[i].slice(0, CHARACTER\_LIMIT));

}

}

displayPosts();

A:

**const** CHARACTER\_LIMIT = 255;

**const** posts = [

"#DeepLearning transforms everything from self-driving cars to language translations. AND it's our new Nanodegree!",

"Within your first week of the VR Developer Nanodegree Program, you'll make your own virtual reality app",

"I just finished @udacity's Front-End Web Developer Nanodegree. Check it out!"

];

// prints posts to the console

function displayPosts() {

for (**let** i = 0; i < posts.length; i++) {

console.log(posts[i].slice(0, CHARACTER\_LIMIT));

}

}

displayPosts();

* ***Const*** the first 2 because you don’t want to reassign or redeclare
* ***Let*** the last one because while you don’t want to redeclare, you want to reassign every time you go through the loop. NOTE: for the for loop, from now on, use ‘let’ and not ‘var’

Q:. slice: what to include, and what not?

A: slice(beginning, excluded\_last)

Ex: slice(0,3)

* Will yield the first, second, third but not the fourth item

**TEMPLATE LITERALS:**

Q: What’s template literals?

A: String literals that include embedded expressions, denoted with backsticks (``)

Ex: `${expression}`

Ex:

const student = {

name: 'Richard Kalehoff',

guardian: 'Mr. Kalehoff'

};

const teacher = {

name: 'Mrs. Wilson',

room: 'N231'

}

* To concatenate the strings, I can use:

let message = **`**${student.name} please see ${teacher.name} in ${teacher.room} to pick up your report card.**`**;

* The ` ` is very important!!! It won’t work unless you have it

🡺 I can drop the quotes along with the ‘+’ or concat. Also can reference the object’s properties inside expressions

Q: If not using template, literals, what is the character for newline?

A: \n

But if I use TL, I don’t need to worry about the \n character:

Ex:

Var note = `${teacher.name},

Please excuse ${student.name}.

He is recovering from the flu.

Thank you,

${student.guardian}`;

\_Can also perform operations, call functions and use loops inside embedded expressions

**DESTRUCTURING:**

**DESTRUCTURING VALUES FROM AN ARRAY**

\_Allows you to specify the elements you want to extract from an array or object on the left side of an assignment

Ex:

const point =[10, 5, -1];

const[x,y,z] = point;

console.log(x,y,z)

* Print 10 5 -1

So for this, the brackets represent the array being destructured and x,y,z represent the variables where you want to store the values from the array.

Also I don’t have to specify the indexes for where to extract the values from b/c they are implied

You can also ignore values when destructuring arrays. Ex: const [x, , z] = point; ignores the y coordinate and discards it

**DESTRUCTURING VALUES FROM AN OBJECT:**

const lolcat = {

yo: 'quartz',

hey: 'rose',

bitch: 21.29

};

const {yo, hey, bitch} = gemstone;

console.log(yo, hey, bitch);

* Will print out: quartz rose 21.29

Set the variables, no matter what name to the object

BUT Destructuring doesn’t have access to “this”:

Ex:

Q: What does this print out?

const circle = {

radius: 10,

color: 'orange',

getArea: function() {

return Math.PI \* this.radius \* this.radius;

},

getCircumference: function() {

return 2 \* Math.PI \* this.radius;

}

};

let {radius, getArea, getCircumference} = circle;

A: NaN because it doesn’t have access to the **this** keyword

Q: What’s the “this” keyword?

A:

Q: Use destructuring to initialize the variables `one`, `two`, and `three`

\* with the colors from the `things` array.

\*/

const things = ['red', 'basketball', 'paperclip', 'green', 'computer', 'earth', 'udacity', 'blue', 'dogs'];

const [one] = things;

const [, , ,two] = things;

const [, , , , , , ,three] = things;

const colors = `List of Colors

1. ${one}

2. ${two}

3. ${three}`;

console.log(colors);

A:

const things = ['red', 'basketball', 'paperclip', 'green', 'computer', 'earth', 'udacity', 'blue', 'dogs'];

const [one] = things;

const [, , ,two] = things;

const [, , , , , , ,three] = things;

const colors = `List of Colors

1. ${one}

2. ${two}

3. ${three}`;

console.log(colors);

**SHORTHAND:**

\_If the properties of an object have the same name of the variables’ names assigned to them, I don’t have to repeat them

Ex:

let type = 'quartz';

let color = 'rose';

let carat = 21.29;

const gemstone = {

type: type,

color: color,

carat: carat

};

console.log(gemstone);

Prints: Object {type: "quartz", color: "rose", carat: 21.29}

This turns into:

let gemstone = {type, color, carat}

\_Also I can omit the keyword function, like below:

calculateWorth: function (){

}

* Turns into: calculateWorth(){ …}

**FOR…OF LOOP**

Q: What are the disadvantages of using the for loop?

A:\_ Having to keep track of the counter and exit condition

\_Good for array, not good for anything else

Q: For… in loop?

A: it eliminates the counting logic and the exit condition, but it’ll print out everything in the array

Ex:

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

for (const index in digits) {

console.log(digits[index]);

}

\_Not recommended for arrays

Q: forEach loop?

A: No way to stop or break

Can only be used with arrays

Q: For… of loop?

A: Used to loop over any data type that is iterable

Ex:

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

for (const digit of digits) {

console.log(digit);

}

+) Can also break a for…of loop any time

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

for (const digit of digits) {

if (digit % 2 === 0) {

continue;

}

console.log(digit);

}

Prints:

1

3

5

7

9

Q: /\*

\* Programming Quiz: Writing a For...of Loop (1-4)

\*/

const days = ['sunday', 'monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday'];

A: const days = ['sunday', 'monday', 'tuesday', 'wednesday', 'thursday', 'friday', 'saturday'];

// your code goes here

for (let day of days){

day = day[0].toUpperCase() + day.slice(1,day.length);

console.log(day);

}

* Use ‘let’ not const cuz u update it repeatedly
* Can’t use template literal cuz it’s gonna appear as “M onday” instead of “Monday”

**SPREAD OPERATOR:**

\_ …

+) used to expand or spread objects into multiple elements

ex: const books = ["Don Quixote", "The Hobbit", "Alice in Wonderland", "Tale of Two Cities"];

console.log(...books);

Prints: Don Quixote The Hobbit Alice in Wonderland Tale of Two Cities

Q: How to concat 2 arrays”

A: 2 ways:

1. Spread operator:

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

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

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

console.log(produce);

Print: [ 'apples', 'bananas', 'pears', 'corn', 'potatoes', 'carrots' ]

1. Concat:

Const produce = fruits.concat(vegetables);

**REST OPERATOR:**

\_Represents an indefinite number of elements in an array

\_Look like the spread operator, but opposite use

\_Spread: represents all and spread them out

\_Rest: putting them back to a package

ex:

**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"]

* First, destructuring the array
* Then “items” is assigned the rest of the values in the array AS AN ARRAY

**Rest Operator + Variadic function**

Q: What’s a Variadic function?

A: A function that takes an indefinite number of argument

Ex:

**function** **sum**() {

**let** total = 0;

**for**(**const** argument **of** arguments) {

total += argument;

}

**return** total;

}

* 1) sum() actually takes an indefinite amount of parameters
* 2) This function uses the arguments object, which represents all the items in the object

I can rewrite this using the rest operator:

function sum(…nums){

let total =0;

for (const num of nums){

total += num;

}

return total;

}

* use for…of..

Q: using the rest parameter, create an average() function that calculates the average of an unlimited amount of number. If there’s no argument, return 0

A:

function average(...nums) {

let total =0;

let avg = 0;

if(nums.length ===0){

return 0;

} else{

for(const num of nums){

total += num;

avg = total / nums.length

}

return avg;

}

}