UNIT 05

LESSON 05.03

JavaScript BOOTCAMP

while loops

do while loops

break

nested for loops

while loop**

A while loop has the essential three ingredients found in a for loop: a counter, a condition and an incrementer. The difference is that in a for loop, these three things are all neatly bundled in parentheses. In a while loop, only the condition is inside the parentheses. The counter is actually declared *outisde* the loop, above it in the global scope; the counter is incremented *inside* the loop's curly braces.

1. Write this while loop:

```
let n = 0;
while (n <= 10) {
    console.log(n);
    n++;
}
console.log(n);
```

do while loop

In a "do while loop", the condition is evaluated only *after* the loop has already run, and so it must run at least once.

2. Write this 'do while' loop:

```
let num = 0;
do {
    console.log(num);
    num++;
} while(num <= 10);
```

Overall, while and do while loops are pretty similar:

- both declare the counter *outside (before)* the loop

- both increment the counter *inside* the loop

3. Write these while and do while loops that concatenate the counter, resulting in output of 1, 12, 123, 1234, etc:

```
    let i = 0, str = "";

    while (i <= 10) {
        str += i;
        i++;
        console.log(str);
    }

    str = "";
    i = 10;

    do {
        str += i;
        i--;
        console.log(str);
    } while (i >= 1);
```

break

Recall that **continue** skips an iteration of a loop. The keyword **break** exits the loop.

4. Write a while loop that ends the loop when the counter gets to 7:

```
    let x = 0;
    while(x <= 10) {
        if(x == 7) {
            console.log('break');
            break;
        }
        console.log('x', x);
        x++;
    }
```

`break` is good for ending a loop once we have found something in an array. After all, if we found what we were looking for, there's no need to check the other items.

5. Write a while loop that iterates an array, looking for a number divisible by 7:

- *if* the number is found, save it to `target` and `break`.
- *if* no number divisible by 7 is found, set the index to -1.
- when it's all over, log the `target` and its index.

```
    let nums = [53, 37, 123, 88, 112, 136, 155];
    let indx = 0, target = 0;
    while(indx <= nums.length) {
        if(nums[indx] % 7 == 0) {
            target = nums[indx];
            console.log('break');
            break;
        }
        indx++;
    }
    if (!target) indx = -1;
    console.log('target', target, 'index', indx);
```

Recall that 0 is falsey, meaning that it returns false in an if-statement. So, just to be fancy, we're using `if(!target)` rather than `if(target == 0)` -- and putting it all on one line with no curly braces.

6. Change the array values so that the loop runs with and without finding a number divisible by 7.

nested loops:

A **nested loop** is a loop-inside-a-loop. Each time the outer loop runs once, it iterates over the entire inner loop. So, if each loop is running ten times, the total is 10x10 = 100 iterations.

7. Run a nested loop where we output the values of both inner and outer counter as we go:

```
    for (let i = 1; i <= 10; i++) {
        for (let j = 1; j <= 10; j++) {
            console.log('i:', i, 'j:', j);
        }
    }
```

This next one involves an array of city abbreviations. Once again, we will use a nested loop. Both loops will iterate over the same array. The inner loop will use the outer and inner counters, **i** and **j**, respectively, to concatenate a pair of cities. The names will be hyphenated, like departures-arrivals: NY-MIA

8. Declare an array of city abbreviations:

```
    const cities = ['CHI', 'NY', 'PHI', 'BOS', 'MIA', 'LA', 'SF'];
```

We will iterate the array with a for loop and console log all possible pairings of cities. We need a nested loop for this, since each city needs to be paired with every other city.

9. Pair each city with every other city--including itself:

```
    for(let i = 0; i < cities.length; i++) {
        for(let j = 0; j < cities.length; j++) {
            console.log(cities[i] + '-' + cities[j]);
        }
    }
```

10. Use conditional logic to only pair cities when `i` and `j` are not equal:

```
    for(let i = 0; i < cities.length-1; i++) {
        for(let j = 0; j < cities.length; j++) {
            if(i != j) { // don't pair a city with itself
                console.log(cities[i] + '-' + cities[j]);
            }
        }
    }
```

What if we didn't want any two-word combo to repeat, not even in the opposite direction. So, MIA-NY means no NY-MIA. For flights this may not be appropriate, because planes fly in both directions, but for some pairings, it may be best to only have the one pairing.

The challenge is to "make smoothies" as pairs of fruits. We want all possible combinations, but without any repeats in reverse, so "banana-peach" is good, but "peach-banana" is bad.

11. Declare an array of fruits:

```
    const fruits = ['apple', 'apricot', 'banana', 'blueberry', 'cherry',
'kiwi', 'mango', 'orange', 'peach', 'pear', 'plum', 'raspberry'];
```

Start with a rough version that gives you all 144 possible pairings--the 12x12 max, including self-pairs like "apple-apple", and revserve pairs like "apple-apricot" and "apricot-apple"). Include a counter that we increment with each pair, so that we can know if the total is going down as we proceed:

12. Make all 144 possible 2-fruit combinations, and keep count:

```
    let counter = 1;

    for(let i = 0; i < fruits.length; i++) {
        for(let j = 0; j < fruits.length; j++) {
            console.log(counter, fruits[i] + '-' + fruits[j]);
            counter++;
        }
    }
```

13. Reset the counter, and run a new loop, where you only get 132 pairs, because the self-pairings ("apple-apple") have been eliminated. This is done with conditional logic: we only make the pair if `i` and `j` are *not* equal:

```
    counter = 1;

    for(let i = 0; i < fruits.length; i++) {
        for(let j = 0; j < fruits.length; j++) {
            if(i != j) { // don't pair a fruit with itself
                console.log(counter, fruits[i] + '-' + fruits[j]);
                counter++;
            }
        }
    }
```

Upgrade again, this time to the final version:

- Each time through the loop, set the just-finished item to "done".
- Add another condition to the if-statement requiring that the current fruit has not yet been 'done'.
- The result is pairs that do not repeat in reverse. We get "apple-banana", but not "banana-apple".
- Save the pairs to a smoothies array, rather than just logging them to the console.

14. Write the final version of the "smoothies maker", in accordance with the above guidelines:

```
    counter = 1;
    const smoothies = [];

    for(let i = 0; i < fruits.length; i++) {
        for(let j = 0; j < fruits.length; j++) {
            // don't pair a fruit with itself or reuse a 'done' fruit
            if(i != j && fruits[j] != 'done') {
                smoothies.push(fruits[i] + '-' + fruits[j]);
                counter++;
            }
        }
        fruits[i] = 'done'; // replace the done fruit with 'done'
    }
    console.log('2-fruit smoothies', smoothies);
```

15. Make it into a function that takes the array as its argument, runs the "smoothie maker" code and, in good input-output fashion, returns the smoothies:

```
    function makeSmoothies(arr) {

        counter = 1;
        const smoothesArr = [];

        for(let i = 0; i < arr.length; i++) {
            for(let j = 0; j < arr.length; j++) {
                // don't pair item w itself or reuse 'done'
                if(i != j && arr[j] != 'done') {
                    smoothesArr.push(arr[i] + '-' + arr[j]);
                    counter++;
                }
            }
            arr[i] = 'done'; // replace done item with 'done'
        }

        return smoothesArr;

    }
```

16. Log fruits to see that all its items have been replaced with 'done':

```
    console.log('fruits', fruits, fruits.length);
```

17. Redeclare fruits under a new name:

```
    const froots = ['apple', 'apricot', 'banana', 'blueberry', 'cherry',
'kiwi', 'mango', 'orange', 'peach', 'pear', 'plum', 'raspberry'];
```

18. Declare an array of tropical fruit, so that we can test the function on more than just one array:

```
    const tropicalFruits = ['banana', 'pineapple', 'kiwi', 'mango',
'mangosteen', 'cherimoya', 'acai', 'noni', 'papaya', 'dragon fruit',
'passion fruit', 'guava'];
```

19. Call the function twice, passing in a different array each time. Save the function call to a variable so that you capture the return value (the smoothies):

```
    let smoothies1 = makeSmoothies(froots);
    console.log('froot smoothies', smoothies1);

    let smoothies2 = makeSmoothies(tropicalFruits);
    console.log('tropical smoothies', smoothies2);
```

- **End Lesson 05.03**
- **Next: Lab 05.03, Lesson: 05.04**