Lesson 05.03 - PROG

while loops and do while loops

break keyword; 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 a while loop, with counter variable `n` declared in the global scope and incremented inside the loop. The loop runs as long as `n` is less than 11:

```
let n = 11;

while (n <= 10) {
    console.log('while loop:', n); // 0, 1, ... 9, 10
    n++; // increment n by 1 each time
}

console.log('after while loop:', n); // 11
```

do while loop

In a "do while loop", the condition is evaluated only after the loop has already run, which means that the loop must run at least once.

2. Write a 'do while' loop with the same counter, condition and increment as in the while loop:

```
n = 0;

do {
    console.log('do while loop:', n);
    n++;
} while(n <= 10);

console.log('after do while loop:', n); // 11
```

Overall, while and do while loops are pretty similar:

- with both, the loop counter is declared in the global scope
- with both, the counter is incremented inside the curly braces

Again, the main difference is that the "do while" loop must run at least once, because the condition is evaluated only after the loop has alredy run once. Let's test this by having a condition that is false from the start, and yet the "do while" still runs once:

3. Write a 'do while' loop with the same counter and increment as in the while loop, but with a condition that is false right from the get go. This loop runs once:

```
n = 0;
do {
    console.log('do while loop:', n);
    n++;
} while(n > 10);
console.log('after do while loop:', n); // 1
```

4. Now, try the same counter, increment and condition with a regular while loop. The condition is evaluated before the loop runs, and so the loop does not run even once:

```
n = 0;
while (n > 10) { // condition is false right from the start
    console.log('while loop:', n); // doesn't run even once
    n++;
}
console.log('after while loop:', n); // 11
```

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

5. Loop the counter from 0-10, but skip the 5 with **continue**:

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

while(i <= 10) {
    i++;
    if(i == 5) continue; // skip the 5
    str += i;
}

console.log(str); // 0 1 2 3 4 6 7 8 9 10
```

6. Reset the variables and do while loop, this time skipping the 6:

```
i = 0, str = "";
do {
    i++;
    if(i == 6) continue; // skip the 6
```

```
    str += i;
} while (i <= 10)
console.log(str); // 0 1 2 3 4 5 7 8 9 10
```

7. Reset the vars and next try a for loop, skipping the 7

```
i = 0, str = "";
for(let i = 0; i <= 10; i++) {
    if(i == 7) continue; // skip the 7
    str += i;
}
console.log(str); // 0 1 2 3 4 5 6 8 9 10
```

The continue keyword skips an iteration, but does not stop the loop. The break keyword, on the other hand, exits the loop.

8. Do a while loop that stops when the counter gets to 5:

```
i = 0, str = "";
while (i <= 10) {
    i++;
    str += i;
    if(i == 5) break; // stop (break) right after 5
    console.log(str);
}
console.log(str); // 0 1 2 3 4 5
```

When should you use a while loop instead of a for loop? When the number of iterations is not specified. In a for loop, the condition and incrementer determine how many times the loop runs. We have done many examples of this, but consider another for reference:

9. Given this array, loop through it to make fruit jellybeans. The loop is set up to run the length of the array, which is 12 in this case, so the number of iterations is specified.

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

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

Now, compare the above to a scenario where the number of iterations is unknown, so for example, what if every time the loop runs we choose a fruit at random. We keep choosing and choosing until we get a 'kiwi'. This could take two tries, or it could take 20 tries. This is the kind of scenario where we want a while loop--

not a for loop. We set up a condition where we keep choosing random fruits as long as (while) the chosen fruit is NOT kiwi.

10. Choose random fruits one at a time on a while loop, until we get a kiwi--then stop. We do not know how many iterations this will take: console.log('\nwhile loop runs until we get a "kiwi"😂;

```
let r = Math.floor(Math.random()*fruits.length);

while(fruits[r] != "kiwi") {
    r = Math.floor(Math.random()*fruits.length);
    console.log(fruits[r]);
}
```

Challenge: Lets try another example where we stop a loop when a target value is found.

11. Write a while loop that iterates the nums array. The loop stops when it finds a number divisible by 7. Log the numbers and their index, as you go, but the last number logged is the first one that is divisible by 7:

```
let nums = [53, 37, 123, 88, 112, 136, 155];

// Challenge solution:
let x = 0, indx = 0;
// As long as (while) the current number is NOT divisible by 7, keep
looping:
while(nums[indx] % 7 != 0) {
    indx++;
    console.log(`Number: ${nums[indx]} - Index: ${indx}`);
}
```

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 100 iterations (10 x 10).

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

```
for(let i = 0; i <= 10; i++) {
    for(let j = 0; 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 iterate over the same array. The inner loop uses both counters to concatenate a pair of cities. The names is hyphenated, departure-arrival style: NY-MIA

13. Given an array of city abbreviations, iterate the array with a nested for loop, pairing each city with every other city:

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

console.log("Includes self pairs (NY-NY)");

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

It works, but each city is paired with itself. Try it again with logic that makes a pair only when `i` and `j` are not equal: console.log("Does NOT include self pairs (NY-NY)");

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

It works, and we have "reciprocal pairs" (NY-MIA, MIA-NY), which makes perfect sense, since flights are in both directions. But in other scenarios we would NOT want reciprocal pairs. For example, if we were making pairs for two-fruit smoothies, we would want mango-pineapple, but pineapple-mango would be redundant.

Using our array of fruits, The challenge is to make all two-fruit smoothie combos but without any reciprocal pairs. // const fruits = ['apple', 'apricot', 'banana', 'blueberry', 'cherry', 'kiwi', 'mango', 'orange', 'peach', 'pear', 'plum', 'raspberry'];

Start with a version that makes all 144 possible pairs. This includes self-pairs like "apple-apple", as well as reciprocal like "apple-apricot" and "apricot-apple". Let's ouput the count, too, so that we know the total:

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

```
let count = 1; // for counting the smoothie combinations
console.log("2-fruit smoothies -- all 144 combos");
for(let i = 0; i < fruits.length; i++) {
    for(let j = 0; j < fruits.length; j++) {
        console.log(`${count}. ${fruits[i]}-${fruits[j]}`);
        count++;
    }
}
```

15. Reset the counter, and run it again, this time eliminating self-pairs:

```
console.log("2-fruit smoothies (NO self-pairs); 132 combox");
counter = 1;
for(let i = 0; i < fruits.length; i++) {
    for(let j = 0; j < fruits.length; j++) {
        if(i != j) {
            console.log(`${count}. ${fruits[i]}-${fruits[j]}`);
            count++;
        }
    }
}
```

16. Let's run it again, this time with logic that eliminates reciprocal pairs. The trick is to start the inner counter (j) each time at one greater than the outer counter (i):

```
console.log("2-fruit smoothies (No reciprocal pairs); 66 combos");
count = 1;
for(let i = 0; i < fruits.length; i++) {
    for(let j = i+1; j < fruits.length; j++) {
        if(i != j) {
            console.log(`${count}. ${fruits[i]}-${fruits[j]}`);
            count++; // increment the counter
        }
    }
}
```

17. Finally refactor the "smoothie maker" as a function that takes a fruits array as its argument, runs the "smoothie maker" code and returns an array of smoothies:

```
function makeSmoothies(arr) {
    const smoothiesArr = [];
    for(let i = 0; i < arr.length; i++) {
        for(let j = i+1; j < arr.length; j++) {
            smoothiesArr.push(`${arr[i]}-${arr[j]}`);
        }
    }
    return smoothiesArr;
}
console.log(makeSmoothies(fruits, fruits.length));
```

Set() data structure

A set is a data structure similar to an array, but with a set there cannot be any duplicate items.

How to Eliminate Duplicate Items from an Array

Make a Set from the Array and then a New Array from the Set If we pass an array containing duplicate values to the Set() object constructor, it returns a Set of unique items -- no duplicates.

18. Pass an array containing duplicates to the Set constructor method. This returns a Set which can have no duplicates:

```
const fruitsSet = new Set(['apple', 'apple', 'banana', 'cherry',
'banana']);

console.log(fruitsSet, typeof(fruitsSet)); // {'apple', 'banana',
'cherry'}
```

Array.from

The array from constructor takes multiple items and returns an array. If we pass it a set, we get an array.

19. Declare a new array set equal to Array.from() with a set as the argument. The result is an array of no duplicate items:

```
const uniqueFruitsArr = Array.from(fruitsSet);
console.log(uniqueFruitsArr); // ['apple', 'banana', 'cherry']
```