CTD Intro Week 2

JavaScript Loops and Arrays

Why Loop?

- Great way to write repetitive code, AKA iteration
  - Look at or change everything in a collection
  - Keep going until the answer is right
  - Do something a variable number of times

Types of Loops

- Iterating through a collection
  - for (let *variable* of *collection)* { … }
    - *variable* is assigned to each item and can be referenced inside the block { … }
    - Runs until everything in the loop has been assigned to *variable*
  - Avoid using for…in to iterate through a collection
    - It has some confusing and unexpected characteristics
- General (standard) for loop
  - for (*initializer ; test ; final-expression) { … }*
    - Typical use: for (let i = 0 ; i < count ; i++) { … }
    - Runs the *initializer* at the beginning
    - Checks *test* at the beginning of each loop and exits the loop if it is false
      - If it is false at the beginning, the loop never runs
    - Runs the *final-expression* after each iteration
- Fun fact: Why does everyone use *i*?
  - Long ago in the FORTRAN language, variables starting with i…m were assumed to be integers by the compiler
  - This naming convention was adopted in later languages like C (without the automatic integer type)
  - It has influenced descendant languages ever since.
    - Usually shorthand for *integer* or *iterator*
  - *It is Important to use meaningful names for variables!*

```
// iterating through a collection
const animals = ['lion', 'tiger', 'bear']
for (const animal of animals) {
    `we have a ${animal} in our zoo.`
}

// a for (general) loop
// computes the square of the sum minus
// the sum of the squares
// of the integers up to max
let sumsq = 0;
let sum = 0;

for (let i = 1 ; i <= max ; i++) {
    sumsq += i * i;
    sum += i;
}
result = (sum * sum) - sumsq;
```

More Loops

- while (*test)* { … }
  - Checks *test* at the beginning of each loop
    - Doesn’t run at all if *test* is false at the beginning
  - Next iteration executes if test is true
  - Initialize any variables used in the *test* expression
- do { … } while (*test)*
  - Similar but the test is at the end of the loop
  - The do { … } block runs at least once

```
// sum of the even fibonacci numbers below max
let total = 0;
let nm2 = 0;
let nm1 = 1;
let fib = 1;
while (fib <= max) {
    nm2 = nm1;
    nm1 = fib;
    if (fib % 2 == 0) {
        total += fib;
    }
    fib = nm1 + nm2;
}

// a do ... while example
// might ask why a for loop isn't used
// in practice do ... while is rarely used
// but handy if you need to test at the end
let num = 1;
do {
  console.log(num);
  num++;
} while (num <= 10);
```

Give me a *break* (and a *continue*)

- Exiting during an iteration: *break*
  - Any time a *break* is executed, the loop exits
    - Code in the loop block which has not run yet is skipped
    - For nested (loop in a loop) cases, execution continues in the enclosing loop
      - It doesn't jump all the way out
- Jumping to the next iteration: *continue*
  - Any time a *continue* is executed, execution continues with the next loop
    - Code in the loop block which has not run yet is skipped

```
// Integer exponentiation
// example of break
function ipow(base, exponent) {
    result = 1; // Initialize result

    while (true) {
        if ((exponent & 1) == 0) {
            result *= base;
        }
        exponent >>= 1;
        if (!exponent)
            break;
        base *= base;
    }
    return result;
}

// example of continue
// log only the positive numbers
const numbers = [1, -2, 3, -4, 5];
for (let number of numbers) {
    if (number < 0) {
        continue;
    }
    console.log(number);
}
```

Nested Loops

- Loops can be used inside loops
  - This can go as deep as you need it to
- Nested loops can be expensive
  - The inner loops runs all the way through for each single iteration of the outer loop.
  - If your code is running slowly, nested loops are a good place to check.
  - Avoid expensive operations in deeply nested loops
- *break* and *continue* only apply to one level of loop
  - Breaking the inside loop doesn't break outer loops

```
// nested loop with break
// What is the smallest positive number that is
// evenly divisible with no remainder
// by all of the numbers from 1 to 20?
let max = 20;
let result = 0;
let broken = false;
for (let i = max ; true ; i++) {
    for (let j = 2 ; j <= max ; j++) {
        if ((i % j) != 0) {
            broken = true;
            break;
        }
    }
    if (broken) {
        broken = false;
    }
    else {
        result = i;
        break;
    }
}
```

Array Basics

- A collection of any type of object indexed by integers
  - Elements are accessed using []
  - First element is 0 – *someArray*[0]
  - Last element is *someArray*[*someArray*.length – 1]
  - .length is an attribute which is one greater than the index of the last element
- Create them using array literals
  - const *myArr* = []; // an empty array
  - Can also use new Array(*size*) // *size* undefined elements
- Accessing an out-of-range element returns undefined

Functional Programming

```
// an example of map
let nums = [1, 2, 3, 4, 5, 6, 7, 8];
let squared = nums.map((num) => num * num);
// squared is [ 1,  4,  9, 16, 25, 36, 49, 64]
// nums is still [1, 2, 3, 4, 5, 6, 7, 8]

// an example of filter
let odd = nums.filter((num) => num % 2 != 0);
// odd is [ 1, 3, 5, 7 ]
// nums is still [1, 2, 3, 4, 5, 6, 7, 8]

// an example of reduce
let product = nums.reduce((acc, cur) => acc * cur, 1);
// product is 40320
// nums is still [1, 2, 3, 4, 5, 6, 7, 8]
```

- Implied iteration
- Applying a function to each element of a collection
- *newArray* = *arrayVar*.map(*function*);
  - *function* is any function which takes 1 variable
  - *newArray* is a new array which contains the result of running *function* on each value in *arrayVar.*
- *filteredArray* = *arrayVar.*filter(*predicate-function*);
  - *predicate-function* is a function of 1 variable which returns true or false
  - *filteredArray* is a new array which contains only the values of *arrayVar* for which *predicate-function* returns true
- *scalarValue = arrayVar.reduce(combiner-function);*
  - *combiner-function takes four arguments, of which the first two are required.*
    - *accumulator: accumulated value, currentValue: current value being processed, [currentIndex], [array]*
  - *scalarValue is the result of applying combiner function to each array element in turn*
- It is common to use the abbreviated function notation for functional programming
  - (*param*) => { … } // an anonymous function which is only used in this one case

Questions?