```
/**
 * recursive functions invoke themselves until a specific condition is reached
 */
function factorial(n) {
    if (n === 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}

function collatz (n, sequence = [n]) {
    if (n === 1) {
        return `Sequence took ${sequence.length} steps. It was ${sequence}`;
    }

    if (n % 2 === 0) {
        n = n / 2;
    } else {
        n = 3 * n + 1;
    }

    return collatz(n, [...sequence, n]);
}

/**
 * call back asynchronous example
 */

function wait(message, callback, seconds) {
    setTimeout(callback, seconds * 1000);
    console.log(message);
}

function selfDestruct() {
    console.log('BOOOM!');
}

wait('This tape will self-destruct in five seconds...', selfDestruct, 5);
console.log('Hmmm, should I accept this mission or not ... ?');

/**
 * using a promise example
 */

const dice = {
    sides: 6,
    roll() {
        return Math.floor(this.sides * Math.random()) + 1;
    }
}
console.log('Before the roll');
let roll = new Promise( (resolve,reject) => { // changed the const to let so it can be
invoked multiple times
    const n = dice.roll();
    if(n > 1){
        setTimeout(()=>{resolve(n)},n*200);
    } else {
        setTimeout(()=>reject(n),n*200);
    }
});
roll.then(result => console.log(`I rolled a ${result}`) )
    .catch(result => console.log(`Drat! ... I rolled a ${result}`) );
console.log('After the roll');

/**
```

```
 * Generalized Functions, using callbacks to write generalized functions rather than having many
 * functions to perform specific tasks
 */
function randomA(a, b = 1) {
    // if a single argument is provided, we need to swap the values of a and b
    if (b === 1) {
        [a,b] = [b,a];
    }
    // random number between a and b or 1 and a if just a single argument was provided
    return Math.floor((b - a + 1) * Math.random()) + a;
}

// adding a callback will allow an additional function to be applied to the random number
function randomB(a,b,callback) {
    if (b === undefined) { // addresses a single argument, assuming the lower limit is 1
        b = a;
        a = 1;
    }
    let result = Math.floor((b - a + 1) * Math.random()) + a;
    if (callback) {
        result = callback(result);
    }
    return result;
}
// callback functions
function square(n) {
    return n * n;
}

function even(n) {
    return 2 * n;
}

/**
 * Functions returning functions can be used to create a generic function that can be changed
 * to meet specific arguments
 */
function greeter(greeting = 'Hello') {
    return function () {
        console.log(greeting);
    }
}

const englishGreeting = greeter();
const frenchGreeting = greeter('Bonjour');
const germanGreeting = greeter('Guten Tag');

/**
 * a closure example
 */
function closure() {
    const a = 1.8;
    const b = 32;
    return c => c * a + b;
}

const toFahrenheit = closure();
// the new function has its own argument but the values of a and b from the original function are still alive
toFahrenheit(30);

function counter(start) {
    let i = start;
    return function () {
```

```
129             return i++;
130         }
131     }
132
133     // this function starts the count using the variable i but returns a function that has
        the ability to change the value of i
134     const count = counter(1);
135
136     count(); // returns 2
137     count(); // returns 3
138
139     /**
140      * Generator example: Fibonacci-style number series
141      */
142     function* fibonacci(a,b) {
143         let [ prev,current ] = [ a,b ]; // initializes the first two numbers based on
            arguments
144         while(true) { // since true is the condition, while will run indefinitely
145             [prev, current] = [current, prev + current]; // everytime next() method is
                called, the code inside loop runs
146             yield current; // the next value is calculated. A special yield keyword returns
                the state of the value
147         }                   // execution is paused until the next() method is called again.
148     }
149
150     // create a generator based on this function, assign a variable to the function
151     const sequence = fibonacci(1,1); // method called next() is inherited
152
153     sequence.next(); // returns 2 (1 + 1)
154     sequence.next(); // returns 3 (1 + 2)
155     sequence.next(); // returns 5 (2 + 3)
156
157     // can also iterate over the generator to invoke it multiple times
158     for (n of sequence) {
159         // stop the sequence after it reaches 100
160         if (n > 10) break;
161         console.log(n);
162     }
163
164     /**
165      * functional programming: pure function example
166      */
167
168     function pureAdd(x, y) {
169         return x + y;
170     }
171
172     /**
173      * using square() to create a hypotenuse() function - pure function example
174      */
175     function hypotenuse(a, b) {
176         return Math.sqrt(square(a) + square(b));
177     }
178
179     function sum(array, callback) {
180         if(callback) {
181             array = array.map(callback);
182         }
183         return array.reduce((a,b) => a + b);
184     }
185
186     // the sum function can be used to produce a mean
187     function mean(array) {
188         return sum(array)/array.length;
189     }
190
```

```
// using the sum, square, and mean functions to build a variance function
function variance(array) {
    return sum(array, square) / array.length - square(mean(array));
}

/**
 * higher-order functions, accept another function as an argument, return another function as
 * a result, or both.
 *
 * Allows generic higher-order functions to be used to return more specific functions based
 * on particular parameters
 */
function multiplier(x, y) {
    if (y === undefined) {   // this allows the function to be curried, otherwise it will return x * y
        return function(z) {
            return x * z;
        }
    } else {
        return x * y;
    }
}

const doubler = multiplier(2); // curried functions
const tripler = multiplier(3);

doubler(10); // returns 20
tripler(10); // returns 30


/**
 * an example of a higher-order function capable of being curried.
 * It expects two arguments but will return another, curried function if only
 * one argument is provided
 * @param x
 * @returns {function(*=): number}
 */
function power(x) {
    return function (power) {
        return Math.pow(x, power);
    }
}

const twoExp = power(2);
twoExp(5); // returns 2 ^ 5 = 32
const tenExp = power(10);
tenExp(6); // returns 10 ^ 6 = 1000000

// we can invoke it with a value instead by using double parentheses
power(3)(5); // returns 3 ^ 5 = 243

/**
 * it is possible to use a curry() function to take any function and allow it to be
 * partially applied
 * @param func  <-- function a an argument
 * @param oldArgs <-- collects all the other arguments together as oldArgs
 * @returns {function(...[*]): *}
 */
function curry(func,...oldArgs) {
    return function (...newArgs) {
        const allArgs = [...oldArgs,...newArgs];
        return func(...allArgs);
    }
}
```

```
254
255  /**
256   * generic divider
257   * @param x
258   * @param y
259   * @returns {number}
260   */
261  const divider = (x,y) => x/y; // returns the quotient of the two arguments
262
263  /**
264   * the divider function with the first argument: 1
265   * @type {function(...[*]): *}
266   */
267  const reciprocal = curry(divider, 1);
268
269  reciprocal(2); // returns 0.5 ( 1/2 )
```