Jellyvision Full Stack Developer Audition

Welcome! This is an open-book audition and you may use any resource at your disposal except for one: biological entities. That is, you can't ask live people to interactively help you. This includes Skype, IRC, or the guy in the other room.

Since you're performing the audition remotely, we have no idea if you're getting human help or not, but we trust that you'll follow the rules since you've gotten this far in the Jellyvision interview process.

Notes:

1. The sections may be worked on in any order you choose. There are no dependencies.
2. For sections that involve writing methods or functions, please also include code that effectively shows the correctness of your methods/functions.
3. You might not finish the audition, and that's ok.
4. If you are able to answer all of the questions before the time limit and you are satisfied with your answers, return your answers early for extra credit.
5. ZIP up and e-mail your finished work to your Jellyvision contact when it is complete.
6. Finally, we thank you for putting up with our exhaustive interview process. We know it can be grueling, but we want to make sure that you're right for Jellyvision, and Jellyvision is right for you. You'll hear from us in a few days about what the next steps will be.

Part 1: Custom Math

This project involves 2 phases. You will be creating a simple JavaScript utility library to work with integers. The library will be partitioned into modules/classes. The module hierarchy should follow an inheritance chain design, with derived modules having an "is a" relationship to the modules they extend. Your methods should build upon other methods in your library.

Phase 1: Create a basic math class that implements the following methods (the utility library only needs to work with integers; we won't even try floating point numbers):

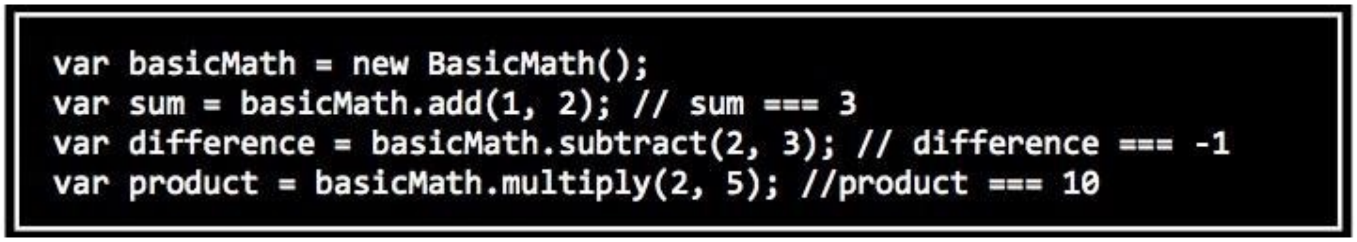
- **add(v1, v2)** - returns the integer sum of adding integers v1 and v2
- **subtract(v1, v2**) - returns the integer difference between v1 and v2 (v2 from v1)
- **multiply(v1, v2)** - returns the integer product of v1 and v2

Phase 1 RULES:

- NO LIBRARIES/FRAMEWORKS ALLOWED, use only plain-old, built-in, vanilla JavaScript.
- **DO NOT USE** built in conversion functions (Number, parseInt, parseFloat, etc)
- You may **ONLY** use **addition** and **unary negation** arithmetic operators, that is "1 + 1" and "-2" (unary negation is NOT subtraction) All other arithmetic operators are off limits; including compound assignment operators.
- Comparison operators are allowed if necessary.
- Functions should be implemented as instance methods
- This class must be named "**BasicMath**".

- Your classes will be defined globally, but you must not create any additional global variables in your library implementation.
- Write your implementations in the file **'custom-math.js'**
- Create another method divide(v1, v2) that returns the integer quotient of v1 divided by v2 (Note: division by 0 should return JavaScript's number constant Infinity)

Example usage:
Your phase 1 class should be able to be used as in the following example:



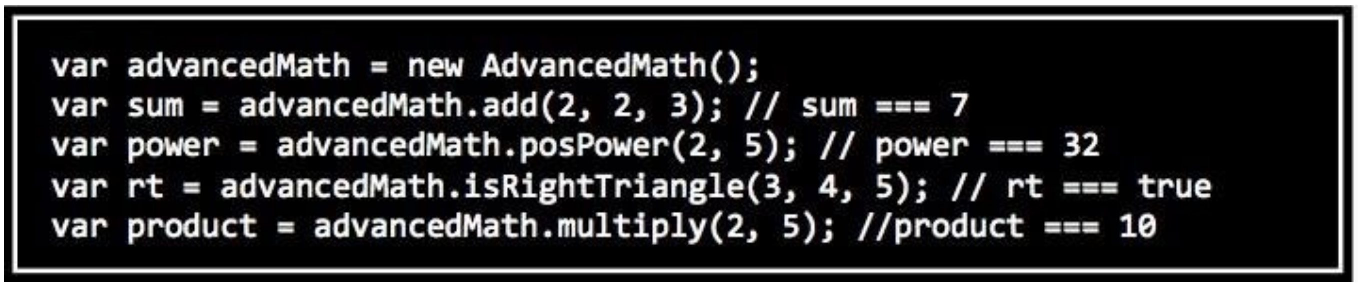
```
var basicMath = new BasicMath();
var sum = basicMath.add(1, 2); // sum === 3
var difference = basicMath.subtract(2, 3); // difference === -1
var product = basicMath.multiply(2, 5); //product === 10
```

Phase 2: Create an advanced math class that extends the basic math class and implements the following methods. (again, these are integer only implementations)

- **posPower(base, exponent)** - returns the integer power of 'base' raised to 'exponent' - you only need to support zero and positive integer exponents. Negative exponents should return zero (0), and exponents of 0 always return positive 1.
- **add(v1, ... , vN)** - returns integer sum of a variable number of integer arguments
- **isRightTriangle(a, b, c)** - given the positive integer lengths of three different sides of a triangle, where c is the hypotenuse side, return boolean 'true' if the triangle is a "right triangle" according to Pythagoras' theorem, and 'false' otherwise. Negative arguments cause a return of false. There are no negative lengths.

Phase 2 RULES:
- Please follow the same rules as in phase 1!
- **DO NOT USE** JavaScript's 'Math.pow', 'Math.sqrt', or trigonometric functions.
- This class must be named "**AdvancedMath**"
- Tip: leverage your previous work.

Example usage:
Your phase 2 class should be able to be used as in the following example



```
var advancedMath = new AdvancedMath();
var sum = advancedMath.add(2, 2, 3); // sum === 7
var power = advancedMath.posPower(2, 5); // power === 32
var rt = advancedMath.isRightTriangle(3, 4, 5); // rt === true
var product = advancedMath.multiply(2, 5); //product === 10
```

Part 2: MySQL

2A: Table Structure

A database system needs to model the following:

Company: The system manages data for many Companies. Each Company has a name attribute.

Each Company also offers one or more Medical Plans to its employees.

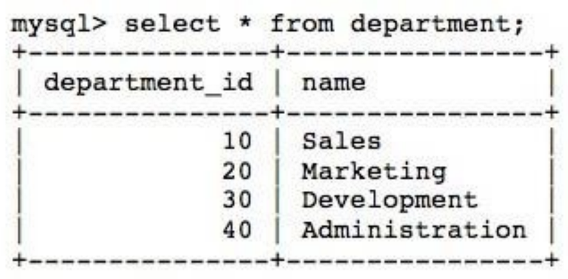
Medical Plan: Each Medical Plan has a name attribute and belongs to exactly one Sponsor. Each Medical Plan is available to one or more Employee Groups.

Employee Group: Each Employee Group has a name attribute and belongs to exactly one Company. Each Employee Group has access to one or more Medical Plans.

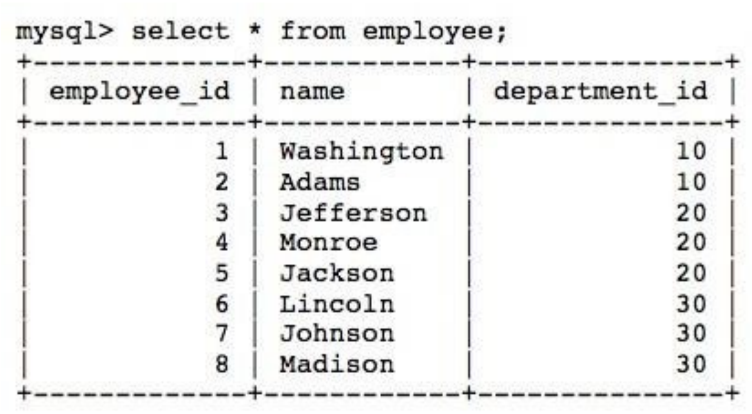
1. Write SQL to create the tables for these entities.
2. Write SQL to retrieve all of the Medical Plans for a given Company.
3. Write SQL to retrieve all of the Employee Groups for a given Company.
4. Write SQL to retrieve all of the Employee Groups who may access a given Medical Plan.
5. Write SQL to retrieve all of the Medical Plans that may be accessed by a given Employee Group.

2B: Writing Queries

The following tables and data are defined in a schema:

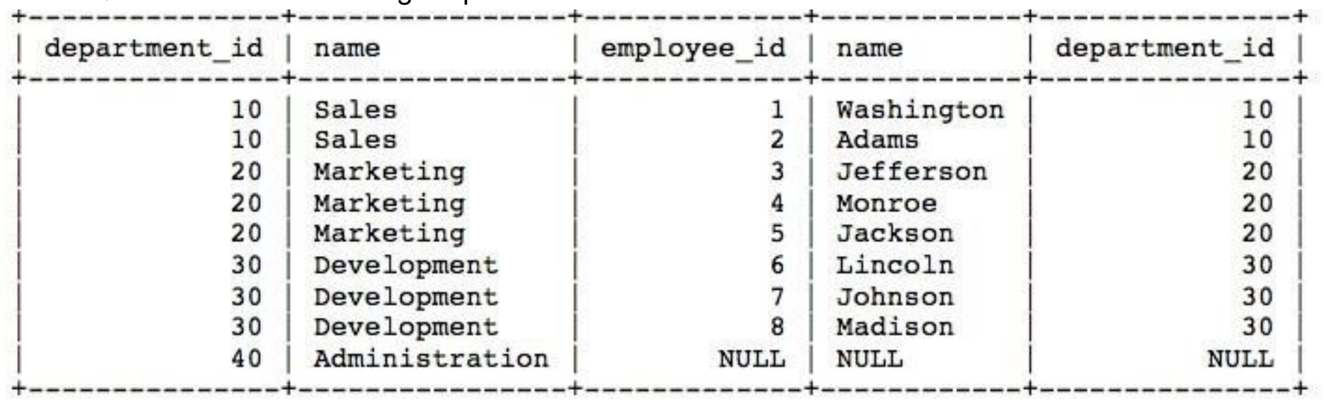


```
mysql> select * from department;
+---------------+----------------+
| department_id | name           |
+---------------+----------------+
|            10 | Sales          |
|            20 | Marketing      |
|            30 | Development    |
|            40 | Administration |
+---------------+----------------+
```



```
mysql> select * from employee;
+-------------+------------+---------------+
| employee_id | name       | department_id |
+-------------+------------+---------------+
|           1 | Washington |            10 |
|           2 | Adams      |            10 |
|           3 | Jefferson  |            20 |
|           4 | Monroe     |            20 |
|           5 | Jackson    |            20 |
|           6 | Lincoln    |            30 |
|           7 | Johnson    |            30 |
|           8 | Madison    |            30 |
+-------------+------------+---------------+
```

Write SQL to return the following output:



```
+---------------+----------------+-------------+------------+---------------+
| department_id | name           | employee_id | name       | department_id |
+---------------+----------------+-------------+------------+---------------+
|            10 | Sales          |           1 | Washington |            10 |
|            10 | Sales          |           2 | Adams      |            10 |
|            20 | Marketing      |           3 | Jefferson  |            20 |
|            20 | Marketing      |           4 | Monroe     |            20 |
|            20 | Marketing      |           5 | Jackson    |            20 |
|            30 | Development    |           6 | Lincoln    |            30 |
|            30 | Development    |           7 | Johnson    |            30 |
|            30 | Development    |           8 | Madison    |            30 |
|            40 | Administration |        NULL | NULL       |          NULL |
+---------------+----------------+-------------+------------+---------------+
```

2C: Optimization

A query is performing more slowly than you would like. What is the first thing you would research when determining how to speed it up?

Part 3: Roman Numerals

The Roman numerals for 1 through 10 are I, II, III, IV, V, VI, VII, VIII, IX, and X. The Roman numerals for 20, 30, 40, 50, 60, 70, 80, 90 and 100 are XX, XXX, XL, L, LX, LXX, LXXX, XC and C. The Roman numeral for any two-digit number can be constructed by concatenating the numeral for its tens and the numeral for its ones.

Examples:

22 is 20 + 2 = "XX" + "II" = "XXII"
47 is 40 + 7 = "XL" + "VII" = "XLVII"
99 is 90 + 9 = "XC" + "IX" = "XCIX"

Using the language of your choice (except JavaScript – we've already seen you write in that language), write a function that accepts a Roman numeral string a returns the integer equivalent of that string.