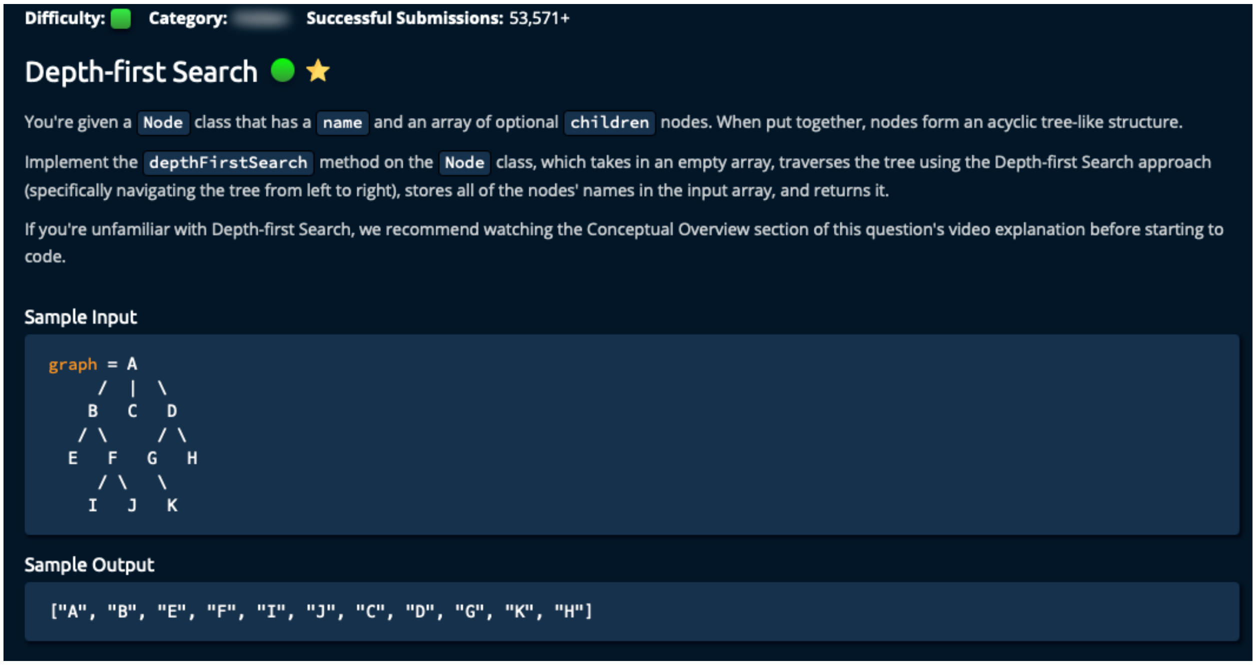
Code Challenge #8 Node Depths (Easy)



Difficulty: 🟩 **Category:** [redacted] **Successful Submissions:** 53,571+

Depth-first Search ● ★

You're given a `Node` class that has a `name` and an array of optional `children` nodes. When put together, nodes form an acyclic tree-like structure.

Implement the `depthFirstSearch` method on the `Node` class, which takes in an empty array, traverses the tree using the Depth-first Search approach (specifically navigating the tree from left to right), stores all of the nodes' names in the input array, and returns it.

If you're unfamiliar with Depth-first Search, we recommend watching the Conceptual Overview section of this question's video explanation before starting to code.

Sample Input

```
graph = A
     /  |  \
    B   C   D
   / \     / \
  E   F   G   H
     / \   \
    I   J   K
```

Sample Output

```
["A", "B", "E", "F", "I", "J", "C", "D", "G", "K", "H"]
```

Solution #1

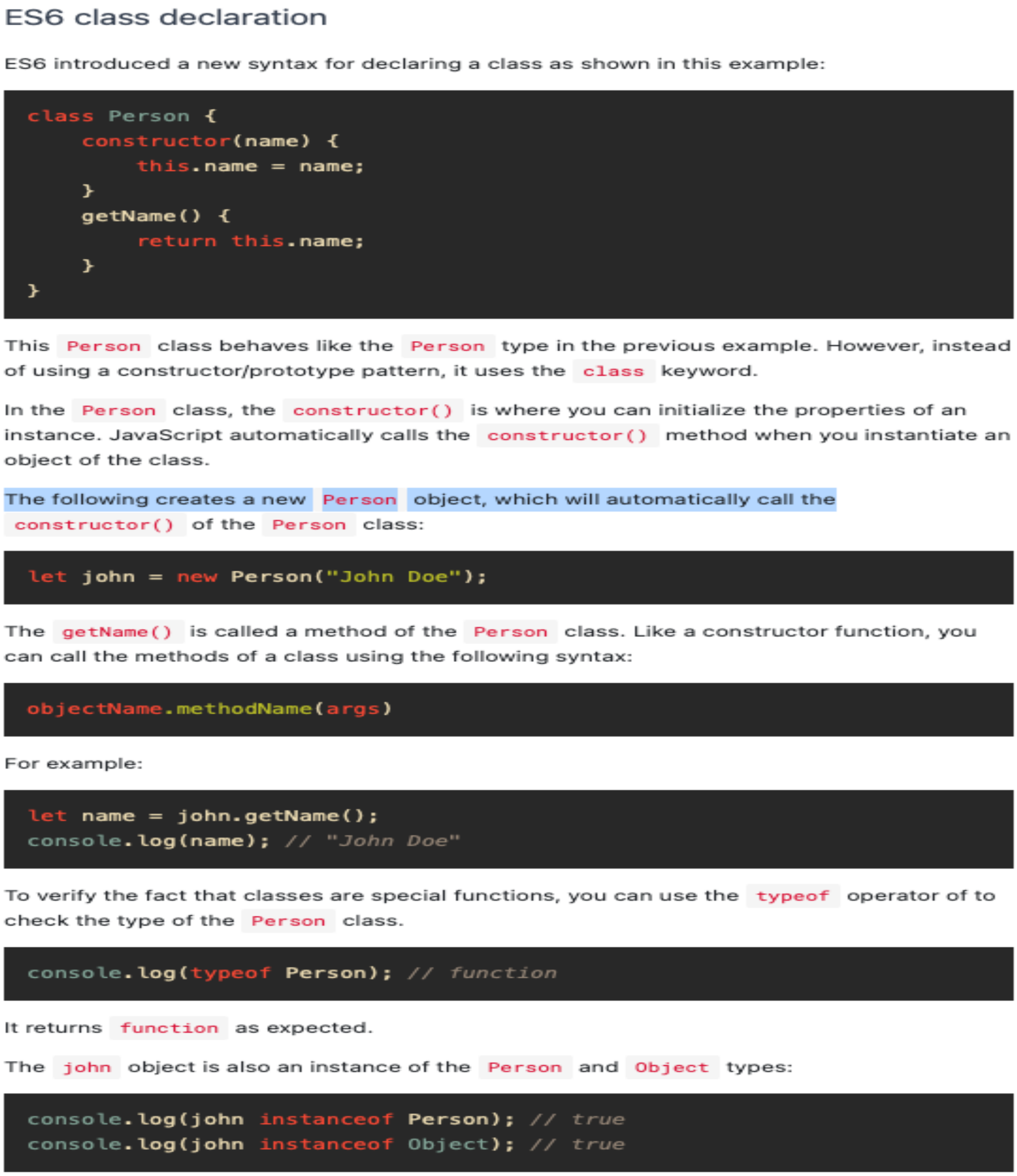
```
class Node {
  constructor(name) {
    this.name = name;
    this.children = [];
  }

  addChild(name) {
    this.children.push(new Node(name));
    return this;
   }

   depthFirstSearch(array) {
     array.push(this.name);
               for (const child of this.children) {
                      child.depthFirstSearch(array)
               }
               return array;
   }
 }
```

Explanation

The explanation for this problem is based on understanding classes in JavaScript. JavaScript classes are different from classes in other programming languages act as syntactic sugar over prototypical inheritance (see here).



ES6 class declaration

ES6 introduced a new syntax for declaring a class as shown in this example:

```
class Person {
    constructor(name) {
        this.name = name;
    }
    getName() {
        return this.name;
    }
}
```

This `Person` class behaves like the `Person` type in the previous example. However, instead of using a constructor/prototype pattern, it uses the `class` keyword.

In the `Person` class, the `constructor()` is where you can initialize the properties of an instance. JavaScript automatically calls the `constructor()` method when you instantiate an object of the class.

The following creates a new `Person` object, which will automatically call the `constructor()` of the `Person` class:

```
let john = new Person("John Doe");
```

The `getName()` is called a method of the `Person` class. Like a constructor function, you can call the methods of a class using the following syntax:

```
objectName.methodName(args)
```

For example:

```
let name = john.getName();
console.log(name); // "John Doe"
```

To verify the fact that classes are special functions, you can use the `typeof` operator of to check the type of the `Person` class.

```
console.log(typeof Person); // function
```

It returns `function` as expected.

The `john` object is also an instance of the `Person` and `Object` types:

```
console.log(john instanceof Person); // true
console.log(john instanceof Object); // true
```

In order to create the node Class we create a Class called Node (always capital for name) and use the constructor function that takes in an argument of name. We then use the this keyword (see here) to assign a property using this.name = name and this.children = [ ]. The second function called addChild takes in an argument of name. This function acts as another set of node with its own children. Within this function we use this.children.push(new Node(name)) and we return this. The final function is depthFirstSearch which takes an argument of an array. Within the function we use the array to push this.name. We then the use for loop using const child of this.children to loop through each using child.depthFirstSearch(array). We finally return array. This function runs in O(v + e) time. V stands for vertex aka nodes and E for edges aka connections between nodes. We iterate through each node plus we go through each edge since we go through children of each node (connections of nodes).