To maintain good engineering practices and standards, we encourage our engineers to document and write comments on their code as much as possible. In the spirit of automation and continuous deployment, we want to automate checks on when the code is merged into the build pipeline.

You have been assigned a story from your technical lead, to create a program that runs every time an engineer checks in code.

The acceptance criteria in the story are:

1) When a file is checked in, scan the file to count the total number of lines.
2) Scan the file to identify comments and count the total lines of comments in the file.
3) After identifying the lines of comments, scan to segregate the total number of single line comments and the total number of multi-line comments.
4) Any line of code that has a trailing comment should be counted both as lines of code and also a comment line.
5) Finally, from all the comments in the file, identify and count the total number of TODOs.
6) Please note, that the file that is being checked in could be any valid program file. Files checked in without an extension can be ignored. You can also ignore file names that start with a '.'.

For the purposes of this story, you only need to focus on creating the program that meets the above acceptance criteria. You do not need to worry about the integration with source control tools. Your program should work by taking in a program file as input and returning an output as follows.

Example:

```
/*
 * Copyright (c) 2017 Capital One Financial Corporation All Rights Reserved.
 *
 * This software contains valuable trade secrets and proprietary information of
 * Capital One and is protected by law. It may not be copied or distributed in
 * any form or medium, disclosed to third parties, reverse engineered or used in
 * any manner without prior written authorization from Capital One.
 *
 *
 */
package hello;

import java.util.Arrays;

import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.Bean;

/*
*
```

```
* @Configuration tags the class as a source of bean definitions for the application
context.
*
* @EnableAutoConfiguration tells Spring Boot to start adding beans based on classpath
settings, other beans, and various property settings.
*
* Normally you would add @EnableWebMvc for a Spring MVC app, but Spring Boot adds it
automatically when it sees spring-webmvc on the classpath.
* This flags the application as a web application and activates key behaviors such as
setting up a DispatcherServlet.
*
* @ComponentScan tells Spring to look for other components, configurations, and
services in the hello package, allowing it to find the controllers.
*
*/

@SpringBootApplication //@SpringBootApplication is a convenience annotation that adds
all of the above.
public class Application {

    // Main method
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    //The CommandLineRunner method is marked as a @Bean and this runs on start up.
    @Bean
    public CommandLineRunner commandLineRunner(ApplicationContext ctx) {

        //TODO: Refactor this code to print out the beans in a sorted order.
        return args -> { //Let's inspect the beans provided by Spring Boot

            System.out.println("Let's inspect the beans provided by Spring Boot:");

            String[] beanNames = ctx.getBeanDefinitionNames();
//            Arrays.sort(beanNames);
            for (String beanName : beanNames) {
                System.out.println(beanName);
            }

        };
    }

}
```

Total # of lines: 60
Total # of comment lines: 28
Total # of single line comments: 6
Total # of comment lines within block comments: 22
Total # of block line comments: 2
Total # of TODO's: 1

Example:

```
/*
 * Copyright (c) 2017 Capital One Financial Corporation All Rights Reserved.
 *
```

```
 * This software contains valuable trade secrets and proprietary information of
 * Capital One and is protected by law. It may not be copied or distributed in
 * any form or medium, disclosed to third parties, reverse engineered or used in
 * any manner without prior written authorization from Capital One.
 *
 *
 */

//Class-based object-oriented programming in Javascript
class Student {
    fullName: string; //Full Name of the student
    // Supports constructor function with a few public fields
    constructor(public firstName: string, public middleInitial: string, public
lastName: string) {
        //TODO: Refactor this to use the ES6 template literal notation
        this.fullName = firstName + " " + middleInitial + " " + lastName;
    }
}

/*
* In TypeScript, two types are compatible if their internal structure is compatible.
* This allows us to implement an interface just by having the shape the interface
requires, without an explicit implements clause.
*/
interface Person { /**/
    firstName: string;
    lastName: string;
}

/*
* Type annotations in TypeScript are lightweight ways to record the intended contract
of the function or variable.
* */
function greeter(person : Person) { //Add a : string type annotation to the 'person'
function argument.
    return "Hello, " + person.firstName + " " + person.lastName;
}

let user = new Student("Jane", "M.", "User");

document.body.innerHTML = greeter(user);
```

Total # of lines: 40
Total # of comment lines: 23
Total # of single line comments: 5
Total # of comment lines within block comments: 18
Total # of block line comments: 4
Total # of TODO's: 1

Example:

```
# Copyright (c) 2017 Capital One Financial Corporation All Rights Reserved.
#
# This software contains valuable trade secrets and proprietary information of
# Capital One and is protected by law. It may not be copied or distributed in
```

```
# any form or medium, disclosed to third parties, reverse engineered or used in
# any manner without prior written authorization from Capital One.

#Author: Ali Bhagat
#This script helps to build a simple stopwatch application using Python's time module.

import time

print('Press ENTER to begin, Press Ctrl + C to stop')
while True:
    try:
        input() # For ENTER. Use raw_input() if you are running python 2.x instead of
input()
        starttime = time.time()
        print('Started')
    except KeyboardInterrupt:
        print('Stopped')
        endtime = time.time()
        print('Total Time:', round(endtime - starttime, 2),'secs')
        break

# Define again() function to ask user if they want to use the calculator again
def again():

    # Take input from user
    calc_again = input('''
                       Do you want to calculate again?
                       Please type Y for YES or N for NO.
                       ''')

    # If user types Y, run the calculate() function
    if calc_again == 'Y':
        calculate()  # Inline comment about the code

    # If user types N, say good-bye to the user and end the program
    elif calc_again == 'N':
        print('See you later.')

    # If user types another key, run the function again
    else:
        again()


def calculate():
    # TODO: Implement the calculate function

def input():
    # TODO: Implement this function
```

```
    return 'Dummy'

def realpath(path):
    '''
    Returns the true, canonical file system path equivalent to the given
    path.
    '''
    # TODO: There may be a more clever way to do this that also handles other,
    # less common file systems.
    return os.path.normpath(normcase(os.path.realpath(path)))
```

Total # of lines: 61
Total # of comment lines: 19
Total # of single line comments: 9
Total # of comment lines within block comments: 10
Total # of block line comments: 3
Total # of TODO's: 3