Murder Mystery

Colin Sullivan, Steven Chen, Ana Paula Centeno

Welcome dear debugging detectives, to a night steeped in mystery. The Mansion awaits, ready for its guests, filled with eerie rooms and items. As the sun sets, the guests enter, and the storm rages on. But unknowingly, not all will leave...

In this assignment, you will investigate a virtual murder mystery by debugging its code. The code simulates a generic board game, in which characters move through a mansion and interact with items while being pursued by a rogue murderer. Completing this assignment will help you develop your skills related to using the debugger and understanding class structures.

Start your assignment early! You need time to understand the assignment code and to become comfortable with the debugger.

To complete this assignment, submit your auto-generated "Answers.out" file

- ○ You do not need to write any code to complete this assignment.
- ○ To generate this Answers.out, simply run the MurderMystery.java driver and answer the questions asked after the game runs.

Overview

It's a dark and stormy night. The wind howls through the trees as lightning flashes across the sky. Through the fog and out of the shadows, it appears...the Mansion.

You are tasked with investigating a murder mystery, by debugging certain aspects of the simulation found within the Mansion class.

- The Mansion class holds a grid of unique rooms, a cast of 6 characters (including one murderer), and a slew of items.
- The rooms, players, items, player decisions, questions, and more are all randomly determined by the loginID entered in the MurderMystery driver. This means that each mystery is unique, and must be solved differently.

When you run MurderMystery.java, you will be asked a series of questions about the nights events. Use the VSCode debugger to find the answers to these questions, enter them via the terminal, and then submit your Answers.out file to the auto-grader.

Do not modify code related to game behavior for the same reason. If you use print statements or write comments, it will shift line numbers -- use the UNedited Mansion.java file for these questions. See the ***Debugging Guide*** *below.*

Implementation

Overview of files provided

- **Mansion.java** is the "game board" where the Murder Mystery takes place.
  - Important attributes are:
    - Room[] roomMap - 2D array of rooms that make up the mansion
    - Person[] players - array of the 6 characters in the game
    - int time - the current time, a multiple of 5. Equals minutes past 6:00pm.
      - i.e. time = 175 is 175 minutes past 6, or 8:55pm.
  - The method nextTurn() runs a single turn of the game (equaling 5 in-game minutes)
    - Information about this method's logic is contained in the code's comments
    - In a turn, players will each have a chance to move and pick up/drop items.
      - The murderer will move last each turn, and attempt to murder the other players.
    - The method will return false if the game has ended, and true otherwise.

You do **not** need to debug any of the following helper classes to solve your questions.

- **MurderMystery.java** is the Driver used to run the game and print the story:
  - When run, the driver will ask for a loginID. Enter **yours**. If it is not your loginID, you will receive a zero.
    - Then it will ask to print a story intro. After, it will print a short ending.
    - It then asks 10 Questions, which you can answer via the terminal. This will generate an Answers.out file you submit to the auto-grader.
  - *See the "How to Run" Section below for information on running the game*
- **Person.java** represents a character in the game.
  - Each character has attributes relating to their name, if the player is alive, current position/dice roll, and more.
  - The Person class has methods that implement movement decisions for players, which is different for innocent and murder.
- **Item.java** represents an item found within the mansion.
  - Every possible room has an associated item. After the mansion is generated, a set number of items will be randomly chosen and placed in their rooms.

    - Each item has attributes relating to its name, whether it's a murder weapon or not marked, what room it belongs to, and more. If an item is used in a murder, it will be marked true, and players will no longer pick up that item.
- **Room.java** represents a room within the mansion.
    - All players start in the same room, "The Foyer"
    - Each room (*except The Foyer*) has a corresponding item that may or may not spawn inside it.
- **Multiple input files** are included, which are used to print out intro/ending text, and generate random player/room/item names. **Do not modify these, as the auto-grader expects them to be untouched.**

- **Answers.out** will be generated in the project directory upon running the **MurderMystery** driver and answering the questions. **"Answers.out" is the only file you submit to Auto-grader**.

    - **DO NOT MODIFY "Answers.out" BEFORE SUBMISSION**
    - **Only use YOUR loginID when running the Driver How**

To Run your Murder Mystery

Running the game:

Run MurderMystery.java via the VSCode run button, either with or without the debugger.

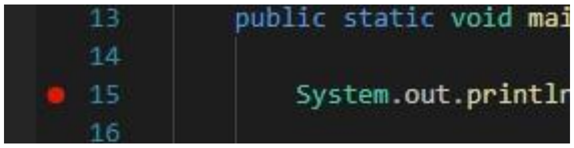
- When run, the driver will ask for a loginID via the terminal. Enter **your loginID**. If it is not your loginID, you will receive a zero.
    - Then it will ask to print a story intro. After, it will instantly simulate the game and print a short ending.
    - It then asks 10 Questions, which you can answer via the terminal. This will generate an Answers.out file you submit to the auto-grader.
        - For "What time/room did ITEM get picked up/dropped?" questions, enter "Untouched" if the item was never touched. For "What time/room did PLAYER die?" questions, enter "Alive" if the player was never murdered.
        - Enter all names exactly as spelled in the Person/Room/Item objects. Enter any times as X:YYpm (no leading zeroes).
- *See the "How to Run" Section below for information on running the game*

Debugging Guide

The first step to debugging is understanding the program that you want to debug. Especially make sure you understand how the Mansion class stores people/items in its code, as well as in its rooms.

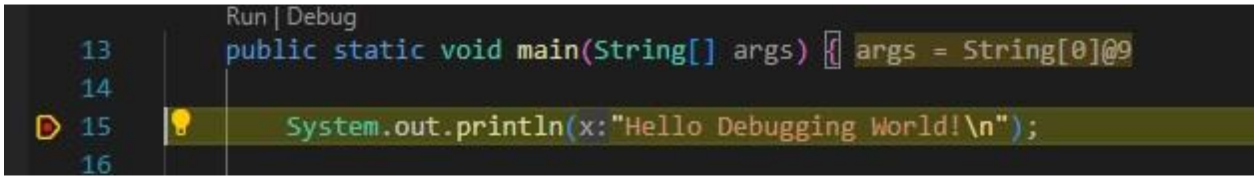
Generally, to debug your program you will set breakpoints in the nextTurn() method in Mansion.java, and use the "Debug Java" feature in VSCode to run in debug mode. VSCode's Debug Mode has the following main features.

- **Setting Breakpoints**:
  - To set a **breakpoint** on a line, click to the left of the line number. A red circle should appear where you click.



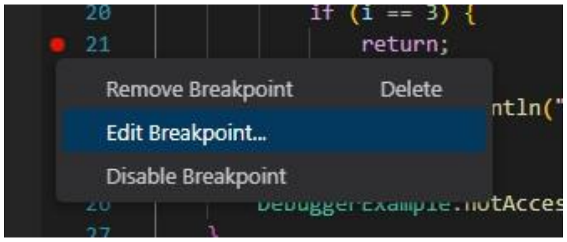
```
    public static void mai

        System.out.printl
```

  - Now, when you run the program in Debug mode, the program will stop before executing that line. It will highlight the line to indicate that it has NOT been ran yet.



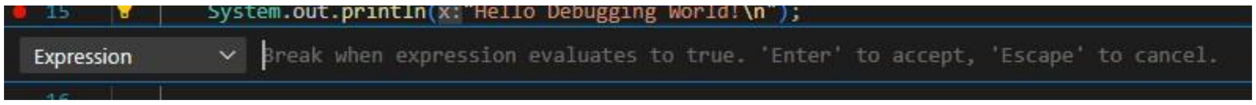
```
    Run | Debug
    public static void main(String[] args) { args = String[0]@9

        System.out.println(x:"Hello Debugging World!\n");
```

- You can also set **Conditional Breakpoints**, which only pause IF the given condition is true at that time.
  - These are VERY useful, and can be used to answer most of you questions! (but are not needed for all questions)
  - To set a conditional breakpoint, right click the line number you want it on, and select “Add Conditional Breakpoint” Or you can right click an existing breakpoint and select “Edit Breakpoint” to turn it into a conditional breakpoint



```
            if (i == 3) {
                return;
                            ntln("
        DebuggerExample.notAcces
    }
```

- Then, you can enter any valid Boolean condition, which will determine when the breakpoint stops.
    - i.e player.getName().equals(“Mrs. White”)
    - i.e droppedItem.getItemName().equals(“Corkscrew”)
    - i.e time == 290



```
System.out.println(x:"Hello Debugging World!\n");
```

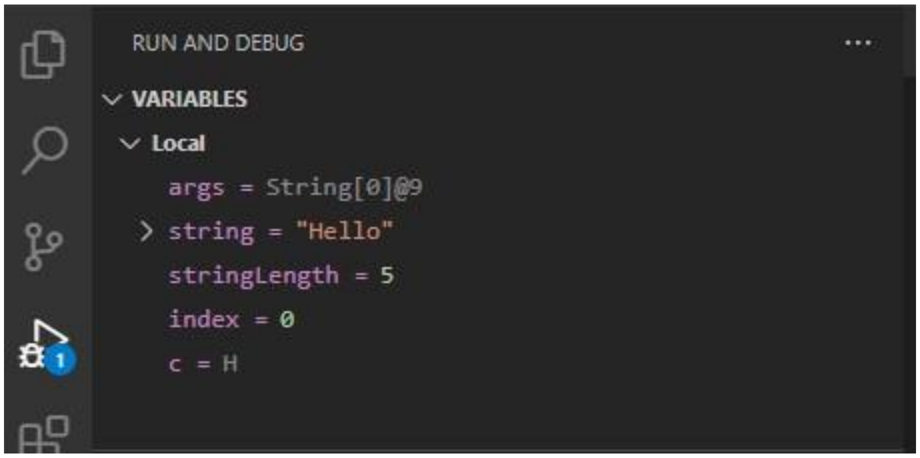
Expression

```
Break when expression evaluates to true. 'Enter' to accept, 'Escape' to cancel.
```

While paused on a line, the debug toolbar will appear with the following options:

- Continue – Continue the program from where it is paused until it hits a breakpoint or the program ends. Also functions as a pause button.
    - Useful to jump between breakpoints, or evaluate the same breakpoint(s) as the program executes them multiple times.
    - This will turn into the Pause button when the program is running, mainly seen in programs with longer or infinite runtimes.
- Step Over – Execute only the current line. Useful to investigate areas directly around your breakpoints
    - Useful to investigate a method's code, by setting a breakpoint before/in areas of interest, and “stepping over” the lines to see how variables change.
    - When encountering a method, this will execute it all instantly instead of line by line, aka “stepping over” them. This is not the case when you are already inside the method.
- Step Into – Enter into the method inside the current line. If no method exists (i.e. int i = 0;) this functions the same as Step Over.
    - Useful to follow the logic of the code as it enters into the method, especially if those methods contain runtime errors.
- Step Out – When inside a method, instantly evaluate the rest of the method and then pause wherever the method returns to.
    - This is useful to quickly exit a large method without stepping through, while still evaluating its return values.
- Restart – If a program is running, terminate the current program execution and start debugging again using the same run configuration.
- Stop – Terminate the current program execution.

The **Variables** window located to the left under the **Run and Debug** menu will show you the values of any variables that are in scope on the paused line. You can use this to stop a program running at a certain point, then view the state of the game via the variable values.



```
args = String[0]@9
> string = "Hello"
stringLength = 5
index = 0
c = H
```

- You can right click any specific variable, and select “Add to Watch”. This will add it to the below “Watch” window, which functions the same as the Variable window. This allows you to pay attention only to specific variable or object values (such as time, a certain item, etc.)
- You can click on variables like “string” above to expand them and show their attributes.
    - This is useful, to see what the eac room in the mansion contains

Experiment with using this window, as it is crucial to good debugging. Understanding how variables and objects are nested is important to finding your solutions!

- DO NOT add code INCLUDING PRINT STATEMENTS to any given classes.

You will need to combine your debugging/detective skills with your knowledge of how the game runs to answer the questions given in the MurderMystery driver.

Formatting Answers

When inputting your answers into the Driver, be careful with formatting.

- Enter names of people/items/rooms **exactly** as spelled in the game.
    - You can reference the room/item/people files in the input folder for spellings
- For “What *time*/*room* did *item* get picked up/dropped?” questions, enter “**Untouched**” if the item was never touched.
- For “What *time*/*room* did *character* die?” questions, enter “**Alive**” if the player was never murdered.
- For time questions, the time field equals minutes past 6:00pm.
    - EX: time = 175 is 175 minutes past 6. So **8:55pm**.
    - EX2: time = 300 is 5 hours past 6. So **11:00pm** (the end).

        - For “Where is ___ at *time*?” questions, check at the **END** of the turn.

Getting Started

1) Run the MurderMystery driver in story mode until the game ends, and **write down your questions somewhere!**

    - Enter all names exactly as spelled in the Person/Room/Item objects and input files. Enter any times as X:YYpm. i.e. 6:35pm or 10:15pm or 8:45pm.
    - Pay attention to which attributes are decided upon mansion generation! Try to answer these first.

2) Locate important lines of code which may relate to your questions.
    - Most of these lines will be within the nextTurn() method in the Mansion.java class. This code related to picking up/dropping items, player movement, and other game actions.

3) Add breakpoints, conditional breakpoints, and print statements. You can answer all your questions in one run if you place breakpoints correctly.

    - Use conditional breakpoints to narrow down **when** you are checking.
    - Combine this with **where** you place the breakpoint to instantly find question answers.
        - i.e. Set a normal breakpoint at the start of the method to check room count, item count, murderer name, and more. Set a breakpoint on the two "return false" statements in nextTurn() to stop at the end of the game.
        - i.e. Set a breakpoint at the .kill() call, with a condition to check if it's killing a certain player.
        - i.e. Set a breakpoint everywhere player.pickUp(item) happens, with a condition for item.getItemName() to equal a certain item name.
        - i.e. Set a breakpoint at the end of the method, to find out where players are/moved.

4) Rerun the MurderMystery driver in story mode and answer the questions, then submit the generated Answers.out file to Auto-grader.

    - You **NEED** to answer the questions for **YOUR** loginID. If you submit an incorrect loginID you will receive a 0.
    - Pay attention to how you spell the names of characters, items, and rooms. It matters for grading. This includes periods and spaces.

Implementation Notes

- **Make sure you have not modified Answers.out. If you submit with a loginID that is not yours in this file, you will receive a zero.**
- You ***MUST*** answer the questions for *your* loginID. If you submit a loginID that is not yours, you will receive a 0.
- DO **NOT** add code **INCLUDING** PRINT STATEMENTS to any given classes.
- DO NOT add/rename the project or package statements.
- DO NOT change the **MurderMystery** class or other helper classes.

VSCode Extensions

You can install VSCode extension packs for Java. Take a look at this tutorial. We suggest:

- Extension Pack for Java
- Project Manager for Java
- Debugger for Java