CSC 232 – Object-Oriented Software Development
Project - Checkpoint 01
Due: Wednesday, October 11th (beginning of class)

This project checkpoint should involve somewhat more programming than previous assignments, so do not waste time getting started. Over the course of several checkpoints, you will be developing a **text-based adventure game** that works similar to the following game called Zork (although yours will not be as complex). When you click on the link above and it opens in a web browser, I would like you to get a sense for how your game should look/feel by performing the following steps:

1. Type the word **help** and press Enter.
   - A text-based adventure game is one in which the game has a prompt (>) that waits for the user to type a command that it knows. The **help** command causes the game to print to the screen a listing of all the commands that the game knows about – as I mentioned, your game will not be as complex as Zork when you are all done but this gives you a sense for (a) what commands are and (b) how to use them
2. Next, type **look** and press Enter
   - Your character is currently at a specific location in the game's world. That location has (a) a **name** ("West of House"), (b) a **description** ("This is an open field west of a white house"), and (c) a **collection of items** currently at this location ("… a boarded front door, a small mailbox, a rubber mat, etc."). In a text-based adventure game, the text serves as the sole means to illustrate the world and its surroundings to the user – it is very much like reading a book in that it engages the user's imagination to see/feel like they are in the world (since there are no visual graphics). Over a series of several checkpoints, **your** game will also have Location objects that are "connected" to adjacent Location objects, a collection of items will be located at each Location, etc.
3. Next, type **examine mailbox** and press Enter
   - This examine command takes a 2nd word as a "parameter" (i.e., examine _____) meaning that we can tell our character to examine a particular item at our location more closely to learn more about the item. In the case of the mailbox, there is nothing particularly special about it upon closer examination. Your game will also support an examine command
4. Next, type **go south** and press Enter
   - This go command also takes a 2nd word (i.e., a direction) as a "parameter" (i.e., go _____) meaning that we can tell our character to travel in a specific direction. In this example, the character traveled south from where they currently were and now have arrived at a new location. This new location has (a) a name ("South of House"), (b) a description ("You are facing the south side of a white house"), and (c) a collection of items currently at this new location ("… no door, all windows are barred"). Later on in future checkpoints (**not** this checkpoint), your game will also be able to support the ability for your character to move from location to location in the world that you create
5. Next, type **go east** and press Enter
   - Once more, we command our character to travel east from the location they currently were at to arrive at a new location ("Behind House"). At our new location behind the house, we find an item (window) that is slightly ajar.
6. Next, type **open window** and press Enter
   - This command causes the character to open the window item and change its "state" from being 'slightly ajar' to now, 'open'

7. Next, type **enter window** and press Enter
    - This command causes the character to enter through the window into a new location, the kitchen. You will notice that in the kitchen location, there are now items (e.g., brown sack and a glass bottle)
8. Next, type **inventory** and press Enter
    - In many games, the character has a personal **inventory** of items that functions like a "backpack" for carrying items around on them as they move from location to location. As you notice, your character right now is not carrying any items ("You are empty handed"). Later on in future checkpoints (**not** this checkpoint) your game will support an inventory for your character to carry items as they travel to different locations
9. Next, type **take bottle** and press Enter
    - This command also takes a 2nd word (i.e., item name) as a "parameter" (i.e., take ____) meaning that we can tell our character to pick up an item at their current location and add it to their inventory ("backpack"). In this example, the character picked up the bottle off the table and added it to its inventory
10. Next, type **inventory** and press Enter
    - Notice how the game now indicates that your character is currently carrying the bottle in its inventory ("backpack")
11. Next, type **look** and press Enter
    - More importantly, when our character looks around in their current location again, we notice that the bottle item is no longer listed as an item found at the location (because it is now in your character's backpack -- type **inventory** so you can verify)
12. Next, type **go east** and press Enter
    - Now, your character is back outside the window and behind the house again
13. Next, type **drop bottle** and press Enter
    - This command also takes a 2nd word (i.e., item name) as a "parameter" (i.e., drop ____) meaning that we can tell our character to drop an item that is currently in our character's inventory at the current location we are in. In this example, the character dropped the bottle from its inventory to the location ("Behind the House")
14. Finally, type **look** and press Enter
    - Notice how the location ("Behind the House") now has the bottle listed as one of its items. Later on in future checkpoints (**not** this checkpoint), your game will support a drop command as well.

I highly encourage you to play around with this Zork game a little bit to get a sense and feel for how I would like **your** game to generally work by the end of the semester. Once you have a good grasp for what a text-based adventure game is and how commands work, you (and your partner) can proceed in completing the following tasks for Checkpoint #1.

Important: Do **not** add more complexity to your game than what I am asking for below (there will be an opportunity for you to "complexify" your game later in the semester, but not now). I will be explicitly grading whether the tasks below have been completed or not. If you do not complete a task that I ask for below, you will lose significant points on this checkpoint.

Task #1 – An Item Class

The first task in building your own text-based adventure game is to create an **Item** class. This class will be used to encapsulate information about an item in the game:

- ☐ Create a Java class named **Item**
- ☐ Add the following member variables to your Item class (recall: these are variables that we feel every Item object should store inside itself):
  - The item's **name** (example only: "Sandwich")
  - The item's **type** (example only: "Food", "Weapon", "Tool", etc.)
  - The item's **description** (example only: "A peanut butter and jelly sandwich")
- ☐ Add a constructor that takes three parameters as inputs: (a) the item name, (b) the item type, and (c) the item description
- ☐ Add an accessor ("getter") method for each of the three member variables listed above
- ☐ Add a mutator ("setter") method for each of the three member variables listed above
- ☐ Add a method with the following exact declaration **public String toString( )** to your Item class. This method should return a String that contains the item's information in the format shown below (note: the example item's member variable values are shown in green while the characters you will want to append to the String are shown in red)

Sandwich [Food] : a peanut butter and jelly andwich

Task #2 – A Location Class

The second task in building your own text-based adventure game is to create a **Location** class. This class will be used to encapsulate information about a single location in the game. If we use our object-oriented perspective, we can imagine that each location in the game is an object that stores its name, description, and a collection of **Items** that are found at that location currently.

- ☐ Create a Java class named **Location**
- ☐ Add the following member variables to your Location class (recall: these are variables that we feel every Location object should store inside itself)
  - The location's **name** (example only: "Kitchen")
  - The location's **description** (example only: "A dark kitchen whose lights are flickering")
  - An **ArrayList** that stores **Item** objects (i.e., the items that are currently at the location) – do not make an ArrayList that stores Strings, it will receive 0% credit
- ☐ Add a constructor that takes **two** parameters as inputs: (a) the location name and (b) the location description. The ArrayList should be constructed as well and initialized to be empty (i.e., no Item objects in it to start with)
- ☐ Add an accessor ("getter") method for the (a) name and (b) description of the location – do **not** add a "getter" for the ArrayList as it will result in a loss of points if implemented and used in your program
- ☐ Add a mutator ("setter") method for the (a) name and (b) description of the location – do **not** add a "setter" for the ArrayList as it will result in a loss of points if implemented and used in your program
- ☐ Add a method named **addItem** that takes a single Item object as a parameter. This method should add the Item object to the location's ArrayList of stored items. Be sure to use the Java API to explore all of the methods that the ArrayList offers.

- ☐ Add a method named **hasItem** that takes a String (i.e., an Item's name that we are searching for) as a parameter. This method should return **true** if the location's ArrayList contains an Item with the same name, otherwise, it should return **false**. You should write this method so that uppercase vs lowercase characters do not matter (i.e., If the user is searching for "Turkey" and there is an Item whose name is "turkey" in the ArrayList, then this method should still return true) – be sure to check out the String API for a String method that you might use. Recall, your Location's ArrayList stores Item objects, not Strings.
- ☐ Add a method named **getItem** that takes a String (i.e., an Item's name that we are searching for) as a parameter. This method should check to see if an Item with that name is in the ArrayList and if so, it should return the matching Item object, otherwise, it should return **null**. You should write this method so that uppercase vs lowercase characters do not matter (i.e., if the user is searching for "Turkey" and there is an Item in the ArrayList whose name is "turkey", then this method should still return that Item) – be sure to check out the String API for a String method that you might use. Recall, your Location's ArrayList stores Item objects, not Strings
- ☐ Add a method named **getItem** that takes an integer (i.e., an index) as a parameter. This method should return the Item object in the Location's ArrayList at that particular index (make sure to check that it is a valid index in your method), otherwise it should return null
- ☐ Add a method named **numItems** that returns how many items are in the location's ArrayList. Note: Be sure to use a good, efficient design for returning this count in order to receive full credit. Use the Java API to explore all of the methods that the ArrayList offers.
- ☐ Add a method named **removeItem** that takes a String (i.e., an Item's name that we are searching for) as a parameter. This method should check to see if an Item with that name is in the ArrayList and if so, it should **remove** and **return** the matching Item object, otherwise, it should return **null**. You should write this method so that uppercase vs lowercase characters do not matter (i.e., if the user is searching for "Turkey" and there is an Item in the ArrayList whose name is "turkey", then this method should still remove and return that Item object) – be sure to check out the String API for a String method that you might use. Recall, your Location's ArrayList stores Item objects, not Strings

Task #3 – A Driver Class

For this checkpoint, your Driver class should **only** contain the main( ) method (however, in a future checkpoint you will add other "helper" methods). The main( ) method should contain an "infinite" loop that continuously prompts the user for the next command and reacts to what they type:

- ☐ Create a Java class named **Driver**
- ☐ Create a static Location variable named **currLocation**
- ☐ Add the **main**( ) method to your Driver class
- ☐ Inside of your main( ) method, you should:
    1. Assign the currLocation variable to "point" at a new Location object that you will use to **test** your commands with for this checkpoint (example: a "Kitchen" Location object)
    2. Add at least **three** Item objects to this Location object that you (I) can use to test your commands with for this checkpoint (example: a Knife object, a Turkey object, and a Plate object)

3. Next, create a Scanner object that reads its data from the standard input stream (System.in). Note: You should only create **one** Scanner object – do not create multiple Scanner objects as that is unnecessary. You will call the Scanner object's nextLine( ) method in an upcoming step below.
4. Enter an "infinite" loop (the player will eventually type the 'quit' command to exit your game)
5. Inside the "infinite" loop, you should **first** prompt the user to enter a command by printing a message to the screen (e.g., "Enter command: ")
6. **Next**, inside your "infinite" loop, you should use the **nextLine( )** method of your Scanner object to (a) wait for the user to type a command and hit Enter, then, (b) this nextLine( ) method will return that line of text (command) as a String. Note: In Visual Studio Code, you type your command where the output text is printed on the Terminal. For example, if you type "examine turkey" as your command in the Terminal, then the nextLine( ) method will return this String ("examine turkey") when you call it. You should store this String to a well-named variable (e.g., String command) as you will need to separate this command into its individual "words" next.

   Important: To keep things simple, you should keep your Item names and Location names to a **single word**. Do not make Item or Location names that consist of more than two words

7. When the user types a command it could be **either** a simple command with only one word (example: "look") **or** it could be a command that has multiple words (example: "examine bottle"). Therefore, we need a way to break the user's command that they type into its individual words. You should look at the **split( )** method in the String class to break the user's command into individual words – see the **1st Example** in the following tutorial for how to split a String into its individual words. The split( ) method returns an array of Strings with each word in its own cell of the returned array. Note: You can assume that I will only put **one** space between each word of the command(s) that I test your program with – you do not need to worry about multiple spaces between words. The reason we need to split a command (example: "examine bottle") into its individual words is so that we can identify (1) the **type of command** ("examine") from (2) its **parameter**(s) ("bottle")

8. Next, your "infinite" loop should use a **switch-case** structure to "jump" to the appropriate case (i.e. command type) for the commands listed below. See the following tutorial for how to use a switch-case structure. You should **not** create a new method like execute( ) as the tutorial does but rather, put the switch-case inside of your "infinite" loop to switch based upon the type of command. Note: You should **not** capitalize your command type in each case as they did in this tutorial (for example, use "examine" instead of "EXAMINE")

9. The commands that your text-based adventure game **must** support for Checkpoint 01 are as follows:
   - ☐ If the user types **quit**, the "infinite" loop should end and the program should exit
   - ☐ If the user types **look**, your program should print out (a) the current location's name, (b) the location's description, and (c) **only the names** of Items at the Location. If you print more than just the Item's names, it will lose points. For example, if I typed the **look** command in the Kitchen, I might see the following:

Kitchen – A dark kitchen whose lights are flickering currently has the following items:
+ Knife
+ Turkey
+ Plate

- ☐ If the user types **examine __NAME__**, your program should try to find the Item object at the current location whose name matches __NAME__ (i.e., the word that the user provided when they typed in the command). For example, the user might type **examine sword** or the user might type **examine book** or the user might type **examine turkey**, in each case, the user provides the name of the item that they wish to examine more about. If the command finds an Item whose name matches what the user typed, then it should print that Item's name and description to the screen using the toString( ) method, otherwise, it should print "Cannot find that item". You **must** assume that the user will type the command all in one "sentence" (i.e., do **not** prompt the user to enter the Item name separately or I will deduct points)
- ☐ If the user types in any other command, your program should print a message such as "I don't know how to do that"

Important: In addition to the code that you add to support the tasks listed above, your program's commands will also be graded on their correctness in handling different scenarios, such as:

- ☐ What if the user just types 'examine' with no Item name?
- ☐ What if the user types 'examine' with an Item that doesn't exist?
- ☐ What if the user types 'examine' with an Item name in uppercase letters instead of lowercase letters? (Note: Uppercase/lowercase should not matter for matching – look at the String class to see if there is a method that will convert Location names, Item names, commands, etc. to all lowercase or all uppercase to ensure that case doesn't matter)