CSC 232 – Object-Oriented Software Development
Project Checkpoint #2

Due: Friday, November 20th (by 11:59PM – Greencastle time)
Extra Credit Opportunity (+**3%**):
If submitted by Wednesday, **November** 18th (**by 11:59PM** – Greencastle time)

In this checkpoint, you will be adding support for multiple locations (i.e., a world) as well as a "**container item**" (i.e., an Item that is capable of storing Items). As a refresher of what you will be building in this checkpoint, open up the game of Zork and type the following commands:

1. Type **look** and hit Enter
   - Recall, this command prints out the Location where your character is currently, its description, and a listing of the Item objects found at this Location
2. Type **go south** and hit Enter
   - For this checkpoint, your first task will be to "connect" Location objects to one another to create a simple world. Our goal is to create a "world graph" (see the following link) where each node (vertex) represents a Location object and each line (edge) represents a "connection" that our character can legally move across to get to a new Location.
3. Type **go east** and hit Enter
   - Notice how your character moves from one Location to the next Location. Each Location has its own set of Items found at that particular Location
4. Type **open window** and hit Enter
5. Type **enter window** and hit Enter
6. Type **go west** and hit Enter
   - You should now be in the Living Room where your character can see a brass lantern on top of the trophy case
7. Type **inventory** and hit Enter
   - An inventory is like the character's backpack that they carry around as they move from Location to Location. Items that are added to the character's backpack at one Location are available for that character when they arrive at a new Location. For this checkpoint, you will be implementing a ContainerItem class and your character's inventory will be a ContainerItem object (i.e., an Item capable of storing Items)
   - Notice: Your character's inventory is currently empty
8. Type **take lantern** and hit Enter
   - This command takes one word as "parameter" (i.e., take ____) and causes the lantern to be removed from the Location and added to our character's inventory. In this checkpoint, you will implement this feature in your text-based adventure game.
9. Type **look** and hit Enter
   - Notice: The lantern is no longer listed as an Item at your character's location in the world (i.e., the Living Room)
10. Type **inventory** and hit Enter
    - Notice: The lantern is now an Item that is in your character's inventory (i.e., backpack)
11. Type **go east** and hit Enter
    - Your character is back in the Kitchen location now where it notices a bottle of water
12. Type **take bottle** and hit Enter
13. Type **look** and hit Enter
    - Notice: The bottle of water is no longer at your character's location in the world (i.e., the Kitchen)

14. Type **inventory** and hit Enter
    - Notice: Your character now has two Item objects in its inventory (i.e., backpack): (1) the lantern and (2) the bottle of water
15. Type **drop lantern** and hit Enter
    - This command takes one word as "parameter" (i.e., drop ____) and causes the lantern to be removed from our character's inventory and added to our Location's list of items. In this checkpoint, you will implement this feature in your text-based adventure game as well.
16. Type **look** and hit Enter
    - Notice: The lantern object is now at your character's location in the world (i.e., the Kitchen)
17. Type **inventory** once more and hit Enter
    - Notice: The lantern is no longer in your character's inventory (because it has been dropped and added to the Kitchen's list of Items)

Once you have a good grasp for how to navigate between locations, take an item from your character's location and add it to your inventory, and drop an item from your character's inventory to your character's location – you (and your partner) can proceed in completing the following tasks for Checkpoint 02.

Task #1 – Modifying your Location Class

The first task outlined below is to modify your existing Location class so that each Location object has the ability to store "pointers" to adjacent Locations. Recall, we want to construct our game's world as a graph (see the following link again). Each Location object will be a node (vertex) that stores connections (edges) to adjacent Location objects that the character can legally move to.

☐ Add a single **HashMap** member variable to your Location class named **connections**
- This HashMap should associate a direction (e.g., "North") to a Location object (i.e., memory address) – therefore, the key type should be **String** and the value type should be **Location**.
- Each Location object will have its own HashMap that maps a String (direction) to a Location object's address. For example, our Kitchen object's HashMap might store the following:

Key	Value
"North" ➔	Hallway object's address
"East" ➔	Living Room object's address
"South" ➔	Front Porch object's address
"West" ➔	Bedroom object's address

Important: These must be direction names (e.g., "north") **not** names of locations – significant points will be lost if the keys are location names instead of direction names

Note: You should be obeying good object-oriented design principles when defining fields -- the connections member variable should be private, not public!

☐ Modify your Location class's constructor so that it constructs/initializes the HashMap

☐ Add a method named **connect** that takes two parameters: (a) a String parameter for the direction's name and (b) a Location object that we want to connect **this** Location to. This method should add an entry into the HashMap that associates the direction (example: "North") to the parameter Location's address. In Task #2 coming up, you will be using the connect method in your Driver class similar to as shown below:

```
Location kitchen = new Location( ... );
Location hallway = new Location( ... );
kitchen.connect( "North" , hallway);
hallway.connect( "South" , kitchen);
```

☐ Add a method named **canMove** that takes a String parameter for the direction name. This method should return **true** if there is a Location object in the HashMap associated with that direction name, otherwise, the method returns **false**. Be sure to review (a) our class slides/video about HashMaps and (b) the HashMap API page to familiarize yourself with what method(s) you can call to implement these features efficiently.

☐ Add a method named **getLocation** that takes a String parameter for the direction name. This method should return the Location object that is associated with the direction, otherwise, it should return **null**

Task #2 – Modify your Driver Class (Part 1)

Next, you will need to modify your Driver class to (a) create your game's world as a graph, (b) start your character at a specific Location object, and (c) test movement between Locations.

☐ Use the static Location variable in your Driver class named **currLocation** that you created in Checkpoint 01 as well as helper function(s) that you will write. **Important:** The currLocation variable will be used to "point" at the Location object where your character is **currently** located. As the user types in commands to move to new locations, this **currLocation** variable should be updated accordingly.

☐ Create a static method named **createWorld** in your Driver class. This method should:

  ☐ Construct **four** new Location objects for your game (e.g., kitchen, hallway, bedroom, etc.)

  ☐ Connect the Location objects together with one another using the **connect**( ) method to form a world "graph" (see the following link as a reminder).
  Tip: Draw your game's world "graph" using paper and pencil first with connections representing arrows from one location to another. **Important**: Remember, when you call connect( ), you are creating a **one-way** connection – not a **two-way** connection. Therefore, if you call **kitchen.connect("west" , bedroom)** ... you may also want to call **bedroom.connect("east", kitchen)** so that your character can travel in both directions and not get "stuck" in a location with no means to escape/move.

  ☐ Add Items to your Locations so that you can thoroughly test that your commands are working correctly. For example, add a "knife" and "turkey" Item to the kitchen and a "lamp" Item to the bedroom. You must add at least **four** different Items spread out across your world's locations so that I will use to test your code. Note: If you put all of the Items in the starting location, I will deduct points if I have to manually add Items to other Locations in order to test your code.

  ☐ Set the **currLocation** static variable to "point" at one of your four locations – this will be where the user's character will start in your world when the game begins.

☐ In the main( ) method, call the **createWorld**( ) method before entering the "infinite" loop to construct the world and setup the game before the user starts playing

- ☐ The commands that your text-based adventure game **must** support at this point are:
  - ☐ If the user types **go** _DIRECTION_ , your program should move the character's current location to the location in the _DIRECTION_ that the user typed (if it exists and is a legal move). For example, if the character is in the kitchen and the user types **go west**, then the character should move from the kitchen to the bedroom if a connection exists to do so. The character should now be able to type **look** and see items that are only found in the bedroom location, not the kitchen.
  - ☐ If the user types **look**, your game should only print Items that are found at that location currently (i.e., the **look** command should work correctly in any location of your world as the character moves around)
  - ☐ If the user types **examine** __NAME__, your game should still correctly allow the user to examine Items found in their current location

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 'go with no direction name?
- ☐ What if the user types 'go' and a direction name that is not North/East/South/West/etc.?
- ☐ What if the user types 'go' and a valid direction name, however, there is no location connected in that direction?

Task #3 – A ContainerItem Class

The third task in your text-based adventure game is to create a **ContainerItem** class. Recall, a ContainerItem is a **special type (i.e., subclass) of** Item with the additional ability of being able to store Items inside of itself (think: chest, vault, toolbox, desk, etc.). In Checkpoint **03** after Thanksgiving break, you will be adding ContainerItem objects to your world, however, for this checkpoint, your character's inventory will be the only ContainerItem:

- ☐ Create a Java class named **ContainerItem** that is a subclass of Item (recall: using inheritance, your ContainerItem class currently has **all** of the fields/methods that your Item class has – so you do not need to re-add/implement those in your ContainerItem class)
- ☐ Add the following fields to your ContainerItem class:
  - An ArrayList named **items** that stores Item objects added to it

  Note: You should be obeying good object-oriented design principles when defining fields

- ☐ Add a constructor that takes three parameters as inputs: (a) the ContainerItem's name, (b) the ContainerItem's type, and (c) the ContainerItem's description. The ContainerItem's ArrayList should be initialized to empty (i.e., no items).
- ☐ Add a method named **addItem** that takes an Item object as a parameter (i.e., the item to be added to this ContainerItem)
- ☐ Add a method named **hasItem** that takes a String (i.e., an Item's name) as a parameter. This method should return **true** if the ContainerItem's ArrayList contains an item with the same name, otherwise, it should return **false**
- ☐ Add a method named **removeItem** that takes a String (i.e., an Item's name) as a parameter. This method should **remove** the Item from the ContainerItem's ArrayList **and** it should also **return** the Item object back when it is finished

- ☐ Override the **toString**( ) method from the Item class so that for ContainerItems, it returns the ContainerItem's (a) name, (b) type, (c) description, **but also** (d) a listing of the Items' names that it contains (note: the example ContainerItem's fields are shown in green while the names of Items it contains are shown in **blue**). If you concatenate **"\n"** to a String, it means "go to a new line" – you may find this helpful to return this String representation that has multiple lines of text. Concatenate the "\n" at the end of the line before you begin to concatenate the next line of text.

  Toolbox [Container] : an old rusted toolbox with drawers that contains:
  + Wrench
  + Soda
  + Screwdriver

Task #4 – Modify your Driver Class (Part 2)

Next, you will need to modify your Driver class once more to support a range of commands related to acquiring and removing items to/from your character's inventory.

- ☐ Create a static ContainerItem variable (field) in your Driver class named **myInventory** that will be able to be accessed in main( ) as well as helper function(s) that you will write. **Important:** This variable will be a special ContainerItem object that will be used solely to store items that your character acquires during the game. Think of myInventory as the character's "backpack" that is carried around as it moves from location to location. You can start your character's inventory with specific Items if you would like **but** your character should be able to pick up items and add them to the inventory ... as well as remove items from the inventory by dropping them in the character's current location.
- ☐ The commands that your text-based adventure game **must** support at this point are:
  - ☐ If the user types **inventory**, your program should print a list of the Item **names** (not Item descriptions) that are currently stored in the character's inventory
  - ☐ If the user types **take** __NAME__, your program should try to find the matching item at the character's current location. If a matching item is found, the item should be removed from the current location and added to the character's inventory (e.g., take _SWORD_). If a matching item is not found, your program should print "Cannot find that item here". You should test this method thoroughly by moving around to multiple Locations and 'take'-ing multiple Items from these different locations.
  - ☐ If the user types **drop** __NAME__, your program should try to find the matching item in the character's inventory. If a matching item is found, the item should be removed from the character's inventory and added to the current location's items (e.g., drop _HAT_). If a matching item is not found, your program should print "Cannot find that item in your inventory." You should test this method thoroughly by moving around to multiple Locations and 'take'-ing multiple Items from these different locations.
  - ☐ If the user types **help**, then your program should print a list of **all** the commands currently supported with a brief one-sentence description for what they do to help a new player understand your game

(see next page for example scenarios that I will use to test/grade your program)

Important: You will want to test each of the commands listed above to make sure that they behave correctly regardless of the user's input. I will be trying to crash/"break" your program by typing in invalid inputs and this criteria will be graded **heavily** for this checkpoint so be sure to test your game thoroughly to earn full credit. Examples of scenarios that your program should be able to support:

- ☐ The '**look**' command should work correctly at any Location (i.e., the look command should only display items that are at your character's current location)
- ☐ The '**examine**' command should (a) notify the user they need to type an item's name if it was omitted and (b) it should not cause the program to crash
- ☐ The '**examine**' command should notify the user that the item does not exist when an invalid Item name is typed (i.e., if I type 'examine shoe' and there is not a shoe item, then I should be alerted that 'shoe' does not exist)
- ☐ The '**examine**' command should work correctly if a valid Item name is typed by printing out information about that item (as required from Checkpoint 01)
- ☐ The '**examine**' command should work correctly if a valid Item name is typed (regardless of uppercase/lowercase letters)
- ☐ The '**go**' command should (a) notify the user that they need to type a direction name if it was omitted and (b) it should not cause the program to crash
- ☐ The '**go**' command should move the character if a valid direction name is provided and a location exists in that direction to move to
- ☐ The '**go**' command should (a) notify the user if a valid direction name was provided but there is no location to move to in that direction
- ☐ The '**go**' command should work correctly regardless of uppercase/lowercase letters
- ☐ The '**go**' command should notify the user if an invalid direction name is used (e.g., go backward)
- ☐ The '**take**' command should notify the user that they need to type an item's name if it was omitted and (b) it should not cause the program to crash
- ☐ The '**take**' command should work correctly if a valid item name was provided at the character's current location
- ☐ The '**take**' command should work correctly regardless of uppercase/lowercase letters
- ☐ The '**take**' command should notify the user that the item does not exist when an invalid Item name is typed (i.e., if I type 'take shoe' and there is not a shoe item at my character's current location)
- ☐ The '**drop**' command should notify the user that they need to type an item's name if it was omitted and (b) it should not cause the program to crash
- ☐ The '**drop** command should work correctly if a valid item name was provided in the character's inventory
- ☐ The '**drop**' command should work correctly regardless of uppercase/lowercase letters
- ☐ The '**drop**' command should notify the user that the item does not exist when an invalid Item name is typed (i.e., if I type 'drop shoe' and there is not a shoe item in my character's inventory)
- ☐ The '**inventory**' command should correctly display the names of item objects that I have taken
- ☐ The '**help**' command should properly display the list of commands currently supported