Assignment 4 Journal

This assignment proved to be an exceptional challenge for me, but an altogether worthwhile one. Throughout the writing of this project, I was required to learn new methods for problem solving, new features of the java programming language, as well as learn new ways to structure and use data. There were times when this assignment almost felt insurmountable, but in the end it has imbued me with greater confidence that I am able to design, plan, and create a larger programming project than I previously thought possible. After writing this assignment, I am excited to continue creating new projects, of increasingly larger scope and ambition.

One of my first tasks in writing this project was researching what exactly the final product might look like. I began by reading some of the history of text-based adventure games, such as ADVENT, and Zork – how they came to be, as well as some of their defining features. Appropriately, for the topic of this project, this became quite a deep rabbit hole to delve into. To my surprise, I found that the text adventure subgenre of gaming was alive and well in the form of internet communities dedicated to 'interactive fiction,' as they called it. I began to delve into archived discussions from old message boards and blog posts, wherein many passionate individuals would espouse on certain aspects of interactive fiction game design – what to do and what not to do, what makes for a good story, and what makes for a boring experience. This was a very interesting experience, as I had no idea that this genre of gaming had any sort of active following. Unfortunately, when it came to advice on actually writing a text-based adventure, these sources were fairly sparse. Most of them recommended using existing game engines that were designed for creating games of this type, with very little discussion on designing text parsers, or creating games from scratch.

Another area of research was that of creating a natural language parser, that was powerful enough to handle user input while not being so complex as to eat up all of my available development time. This proved challenging, as searching for language parsers was in itself a deep topic, one that would be worth it's very own, focused project. Most of the discussions and articles I found on the topic were far too broad to be of use for my project, concerning the creation of parsers capable of handling real-world human interaction, while providing meaningful responses. All of these ideas gave me some notion of what my final parser should look like, but I could see that they were far too in depth for what I required.

I set out to create a parser that was capable of filtering fairly short strings of user input into a meaningful pair of words, which I dubbed verb-object pairs. The idea was to have my game filter down user input until it was distilled to this pair of word, and then have the game direct execution of commands based on the verb word, while the object word provided context, allowing a single verb function to provide a multitude of outputs. This proved to be a relatively easy task, as I was able to filter words based on the legal verbs that the game could understand, and then also only select words that were meaningful to the game as objects. I determined to assign word meaning through the creation of word lists that corresponded to all legal verbs, and all legal objects – that is, objects that exist within the game, and verbs that correspond to game methods – and to assign to each of these legal words several synonyms that I thought were likely to be used by a player of the game. In order to assign synonyms, I created a mapping, using hash maps, with the key of the map being the 'base' word, and the associated elements, held in string arrays, as the synonyms. This way, I could search through the map and, if a word matched one of the elements or keys of a hash map, return the key that corresponded to it, yielding the base word. This approach worked very well, and I am happy with the result. The only additional feature that was required for this parser was to add a third element to these word pairs – making them word triplets – that acted as a context-saving element of the resultant string array. This context word was then used to provide context-sensitive feedback when a command did not work, or the user attempted to do something to an object that

was not in fact a game object. This allows the game to more gracefully handle cases where the user inputs commands that do not fall within the confines of legal words.

The next challenge was determining how to store game data in text files. This proved quite difficult to me, and I explored a couple of options before finally settling on my final implementation. At first, I thought to use JSON files as my data storing method, but after some consideration thought that they weren't the optimal fit for the job. Although a JSON file would accomplish the task admirably, I was unsure whether or not using them would cause issues with compiling and running my game on machines other than my own. Since it was unclear what packages would be available on my target machines – that of my marker, or anyone else who chooses to compile and run the game – I determined to create a novel structure of my own design. Having looked at the JSON structure a bit in my research, I determined to create a structure that served as a 'poor man's' JSON – that is, a simplified version of the format. I decided to organize my data into fields, denoted by the fields name followed by a space, followed by a pair of curly braces inside of which all of the properties of the field would be placed. Each property was delimited by semi-colons, in order to allow for spaces and other punctuation to be used in the actual values of a given property. Some properties were further delimited by spaces, slashes, or commas, as appropriate. I nearly gave up on this approach, as parsing this style of text proved to be quite challenging for me. However, after some research on regular expressions, and how to use them in Java, I began to forge a decent data parser, which I added to my Parse class. After a decent amount of tweaking, and playing with the regex in my matcher (and, admittedly, a few choice words directed towards my computer) I had a simple data parser that could search a text file and pull from it specific data, as called for.

With the user input parser, and data parser in place, I felt that the majority of the challenge of this assignment was behind me. I then made a rough mapping of what I believed each class, as outlined in the assignment requirements document, would accomplish in my program. I determined to have an Inventory, Item, Control, Character, Events, Game, and Location classes to begin with. The Inventory class would be used in the Character, and Location classes, as well as in Control to manage the available items in a Location, held by a Character, or held by the player character. Originally I wanted to have the player character be a specific instance of the Character class, but I decided that it would be easier to implement if all of the player variables were global static variable belonging to the Control class. The Location class was relegated to keeping track of all the things that make a location a location. This included the items that are available there, the characters present there, as well as descriptions of the room and any events that could happen. The Item class was similarly responsible for all of the properties that make up an item, such as it's name, description, associating any related events to it, and so forth. The Control class would act as a traffic controller, calling on other classes, and directing user input received from the parser to call on the appropriate methods for a given command. With this road map outlined, I began developing all of the classes in detail.

I began with the Inventory class, as it was likely the most simple of all the classes. It is essentially a wrapper for an ArrayList, holding Item objects. I created several overloaded constructors that could receive as input nothing – corresponding to a new, empty inventory – a list of items, or a data path and associated field, which would allow instantiating an inventory object replete with all of it's items. Additionally I furnished this class with methods for searching through an inventory, at first by passing an Item object and searching for it, and later by passing the name of the desired item as a string, and comparing that string to the names of the item objects held by the inventory. The searching methods made me realize that searching for an item could be difficult, as passing a variable that holds a reference to an Item object could cause the equals method to not function properly, if those two item objects were not the actual same object. After some research, I found that I could override the equals method for Items, as well as the hashcode method in order to change how equality was expressed. I found the skeleton of the code I needed to implement this

functionality on Baeldung.com. I chose to define item equality as any two items that share the same exact name with one another – thus I could pass new item objects into these methods in order to search for an existing item object bearing the same name. This proved to solve a lot of the issues I was having in finding a particular item in the list. Finally, I added some getters to retrieve the various item properties that other parts of the program may need, such as the description and name of the object.

Next came the Character class, which was much the same as implementing Inventory, at least at first. I began with basic properties that would make a character object unique, such as the name, description, the description that would be printed upon looking at a character more than once, an array to hold dialogue options, and their character inventory. Similarly to the Item class, I overrode the equals and hashcode methods in order to define equality, to aid in searching for and comparing objects in other parts of the game. Later, once I was using the character class in other areas, I would add a few more properties, such as a boolean toggle that indicated whether or not the character had had their associated event (if they had an event associated with them) triggered by the game at any point. This way, I could tie character events to certain repeatable actions used for interacting with the character without having an event trigger more than once if I didn't want it to. I also realized I needed to add a method to track how many times the character had been spoken to, as that was the way in which character events were triggered, as well as what determined which dialogue option was printed upon speaking to that character.

The Game class is generally quite simple, as it's purpose is simply to execute the 'core loop' of the game. This is essentially an infinite loop that will execute as long as the game is running. It calls for user input, prints out the prompt for said input, and passes this input first to the parser, and then to the Control class for execution. Once the various classes have completed their tasks, they exit their particular methods, allowing the game to advance to another cycle, causing the Game to repeat the prompts and passing of information between classes. Because it's job is to simply coordinate the other classes at the most basic level, this is the simplest and shortest class, in terms of code.

Location was the next class on the itinerary. This proved to be a more difficult class to implement than I initially expected, as locations are at the core of the game. The majority of the interaction the player does is directed towards a location in some way or other. Similarly to the Item, and Character classes, the Location class had basic properties consisting of a name, description, return description, inventory of items, list of characters, and so forth. What was more complex, was that this was where I decided to store the 'game map' which was to be loaded when the game was initialized. In order to create this map, I first declared an array list to be filled with Location objects, and stored it as a static variable in the Location class. I then created a generateMap method, which would take the starting location as a seed, and then generate all of the location objects connected to it by reading all of the names of the connected locations from the locations data file. It would then recursively do the same thing for every location that it created. This proved to be a tricky problem, as I immediately caused an infinite loop to occur, as this method would find locations that had already been mapped in the list of locations for a currently-being-mapped location, and then try and map those locations, and so on ad infinitum. Although I had thought of this when I was initially implementing my mapping algorithm, this problem persisted. I was quite perplexed as to how this could be happening, and eventually made a post on the learnjava subreddit on Reddit to ask for help. I posted my problem code, as well as my override of the equals and hashcode methods only to have it pointed out that I was using the wrong comparator when comparing names in the equals method. I had mistakenly used the '==' comparator, rather than the equals method when comparing the name strings of the two objects in this method. Quite a foolish mistake, but one that I had made in all of my overrides, but hadn't caught yet because the game was still taking shape. After I had fixed that issue, the mapping algorithm worked beautifully, and I was well under way to having a working prototype of a game.

With Character, Item, Location, and Inventory implemented, I began to work on Control, in order to bring it all together. The centerpiece of this class is the dispatcher method, which acts as the traffic controller for the entire game. The parser generates the word-object-context triplet from user input before being passed to dispatcher, which is essentially a single large switch statement. This switch statement uses the verb of the aforementioned triplet as the toggle variable, which directs the program to activate the appropriate methods. For instance, if the player enters ‘go north’ the triplet would be [go, north, null]. The dispatcher method then funnels this to the go method which then distills the ‘north’ context into a game direction, allowing the game to search for the attached location, if it exists, and move the game context into that location. This means printing the appropriate description of the location, and checking if the thief character appears and tries to steal the jewel from the player. With dispatcher in place, I implemented all of the methods associated with the various core verbs, and with that I had enough of a game built to begin testing it. I created some small sample text files in order to test loading data, and moving between rooms, picking up items, and talking to characters. Most of these methods worked well enough with a few tweaks, but once I began to scale things up to a more complete game I had some difficulties.

One of my major difficulties in scaling the game up to full size was organizing which classes should keep track of which game properties. As I added more and more features, I quickly found myself having difficulties keeping track of which classes held which data at which times. This is a definite place that more planning and diligent design would have saved me a lot of problems. For future projects, I will certainly be adding a data portion to my class design, as it will allow me to keep things straight when I get to the point of having my classes interact with one another. An example of this growing pain is where I decided to keep the global game-state variables. At first, I thought to keep all of these variables in the Game class, which was responsible for executing the core loop that made up the game, and calling on the other classes at appropriate times to have the game actually execute. In the end, however, I realized that I was constantly calling for game state variables from the Game class in the Control class, and determined that if I was going to be using this data primarily in Control, that I should simply store the data in that class as well. This proved to help immensely with ensuring that the data I needed was easy to access. It also allowed Control methods to change global state variables more easily, allowing a less complicated update process. This avoided needing to constantly call on methods to update global variables in order to keep all of the disparate parts of the program in the same context.

As I added more pieces to the puzzle and expanded Control, I began to realize that other portions of the program needed to be more robust to provide sufficient information to the Control class in order for Control to properly execute what was required of it. This included adding the aforementioned context element in the parse output, as well as making the parsing of objects and their synonyms more robust. Rather than having the object parsing methods simply operate on an immutable list of game objects (which would have been prohibitively long to declare within the program in a stylistic way), the parser needed to both pull the object name, and associate synonyms with it in one pass. This proved to be fairly easy, using the general data parser as a template – with a little bit of a change in how the regex was capturing groups, I was able to pull both pieces of data in one go. Another instance of this, was realizing that for methods that required a game direction, using strings was difficult. Instead, I created a new class, Direction, which consisted of a single enum declaration that defined all of the legal directions available in the game. I then created a hash map in control that associated game directions with locations, and used another switching method, similar to dispatcher, to associate location names to their direction dependent on their index in the location name array yielded from the data file. This made the go method particularly easier to implement, and cut down on a lot of complexity.

With the bones of the program in place, I began implementing the Events class, which would serve as a catch-all class that implements any special game behaviours not covered by the other game classes. This consisted mainly of the creation of several mini-games that would serve as small side tracks and challenges, at the end of which the player would receive important game items. At the core of this class is the execute method, which is another near copy of the dispatcher method found in Control. It receives as input the name of an event, and then associates that name with a particular event method, for execution. This was easy to implement, and gave me few problems. Similarly, writing the event methods proved quite easy. The difficulty in implementing events ended up being outside of the event class itself, in that it became clear that I did not know how or when to check for or call a particular event during game execution. Initially I thought to have a method check for relevant events at the beginning or end of every game cycle. That is, as one of the steps in the core game loop, have the game check for the existence of events on every game object present in the current context, and execute those events if they are relevant. This proved to be problematic, because I found that if an event became relevant, it would be called upon again and again, as the context could not change during an event, and so the event would never become irrelevant again. This approach may have been feasible, given enough attention and time, but I determined to try and find a more elegant solution. I am not sure the solution I found ended up being much more elegant, but I ended up classifying my events according to what could trigger them. For character events, I wanted them to trigger after all of a character's dialogue was exhausted, so I added a check to the talk method to determine when the character was out of dialogue, and to search for events associated to that character and execute them. Similarly, the touch method checks items and characters for events and executes any that it finds. The go method has a persistent check, where it randomly assigns an integer a value between 0 and 9, and if the integer is 0, it adds the thief character to the room and executes the steal method – stealing the players jewel, if they have picked it up. I toyed with the idea of having the look method activate events, but in the end decided that it would be too surprising and potentially punishing to cause events to happen to the character just by observing their surroundings.

While developing the Events class, I determined that I wanted to have the ability to lock (and unlock) doors, as well as have doors that were too big or too small for the player to pass through. This would allow me to have the 'drink me' and 'eat me' items change the player's size for a meaningful reason, as well as have a golden key item that is featured in the Alice in Wonderland story. I realized that I had no way of tracking the status of an exit for a location, and so I had no way of locking a door, or making a door too big or too small. This caused me to refactor locations a little bit, adding a new property to the location data fields. This new property was simply a list of six strings which could be of specific words that corresponded to different door statuses; these were OPEN, LOCKED, BIG, SMALL, and LOST. Open allowed for free movement through the door, locked corresponded to a locked door in the game, allowing the player to open a shortcut, big and small denoted the size of the door, which must match the size of the player to permit passage, and lost would trigger the forest maze mini game – another way to trigger events with the go method. These statuses were checked every time the go method was called, and would stop the player from moving between locations if they were not satisfied. This added a great deal of functionality and interactivity to the game, and I am quite proud of their inclusion.

Now that I had another layer of complexity implemented, I took to testing the game with dummy locations again, in order to iron out any new bugs that had cropped up. This was tedious, but not difficult, and once I was fairly confident everything was working how I wanted it to, I began writing the full game's data.

In order to write the full game data, I first drew out a game map, and outlined what I wanted to have in each location. In retrospect, I should have done this earlier in the process, as it ended up serving as a feature checklist for events I wanted to implement, as well as what exactly a location was

required to do and be. I made a list of all of the characters I wanted to represent within the game, as well as what items I wanted in the game – ensuring that I met all project requirements in doing so. This step was heavily influenced by what I had already implemented in the project – more so than I initially would have expected it to. I found myself designing my game story based on the capabilities of the game engine I had written, rather than the other way around. Should I do this project again, I think I would have preferred to have a fully fleshed out game idea, and then write the engine to accommodate what I wanted to achieve. That being said, being forced to find ways to fit what I wanted to do within the 'lines' of what I had already accomplished was an interesting challenge in it's own right, and one that ended up being a fun game of sorts. I found adhering to my own game data structures to be relatively difficult, but I had thankfully documented all of the property ordering for each data field, and simply needed to continuously refer to this documentation in order to ensure things were in the proper order for the parser to understand.

Much to my relief, the number of problems from this point on were minimal. I had a few strange bugs that took some time to track down, such as misnaming certain items, characters, or other objects in the data files themselves causing exceptions that I didn't quite understand at first, but beyond that the engine functioned mostly as I had hoped it would. With my data written, I began to play the game and make notes of things that I wished to change or tweak. After a few rounds of this, I arrived at what is now the final version of the game. Most of the final edits were correcting various formatting issues that I wasn't happy with. Although there are still a few minor tweaks and polishing steps that I could perhaps continue to work on, I believe that the game is in a very strong state at present. At some point, one needs to stop polishing, or be caught in a perpetual state of perfecting something.

The largest lesson I am taking from this assignment is that it pays to have a more thorough plan in place before beginning to write code, and to have a more robust method for testing code being actively written. I had several issues where I had a hard time tracking down bugs because I had written a large portion of the program before I had tested any part of what I had just written. In the future, I would like to implement unit tests for each method that I create in order to ensure that each method is at least behaving in the manner I expect it to, rather than finding issues with it long enough down the line that I have a hard time finding the issue at all. Similarly, having a more complete road map for what a finished game looks like would help serve as a checklist for features I plan to implement, rather than simply implementing items as they come to my mind, without thinking about how they fit and function as a smaller part of the whole program.

This project has been an excellent overall experience, and has shown me that creating larger pieces of software is attainable. I hope to use the lessons learned here to go on to create more personal projects of increasingly greater scope in order to further my abilities as a programmer. I think I will be chasing this feeling of accomplishment for some time now.