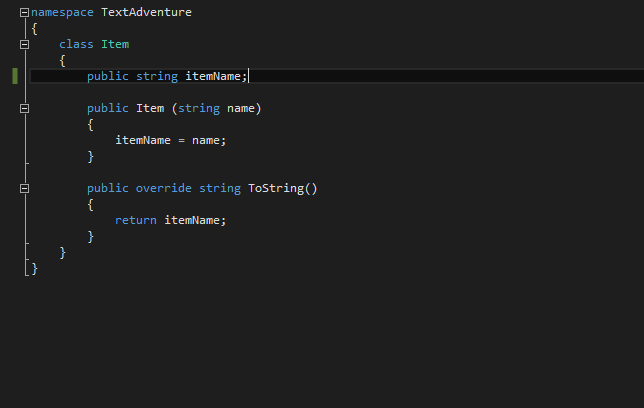
Research and Code Documentation

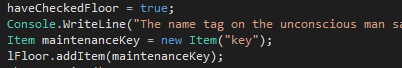
# Item Class

To create the item class, I used what is known as a constructor, specifically and Instance Constructor. This is essentially an empty class that provides the barebones tools to create an instance of the class in other parts of the code, allowing any variables to be handled outside the class code block (akin to instantiating a game object in Unity). Given that a game like this can have many items, it stands to reason that it can have many instances of the Item class.



*Above: A screenshot of the Item class, in its entirety*

The entire item class is made up of these two blocks. Because the instance of the item, and all code handling that instance is declared in the Game class, the Item class can get away with having a single variable (“itemName”). This variable is assigned to during instance declaration, using the parameter “name” to parse the name of the item into the itemName variable.



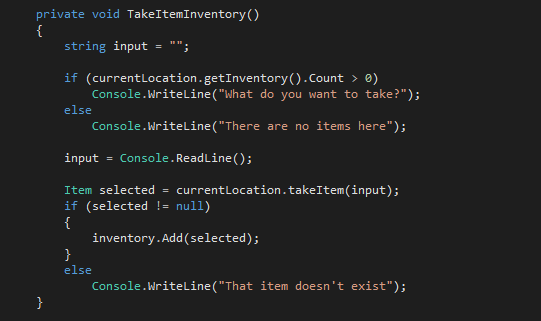
*Above: An example of Item declaration in the Game class, showing the use of a parameter to parse in a string*

The Item class also contains an example of an override. This is a way of modifying the behaviour of an existing method. In this example, when Item.ToString() is called, the program returns the name of the instance, rather than the class name (e.g. “key” rather than “TextAdventure.Item”).

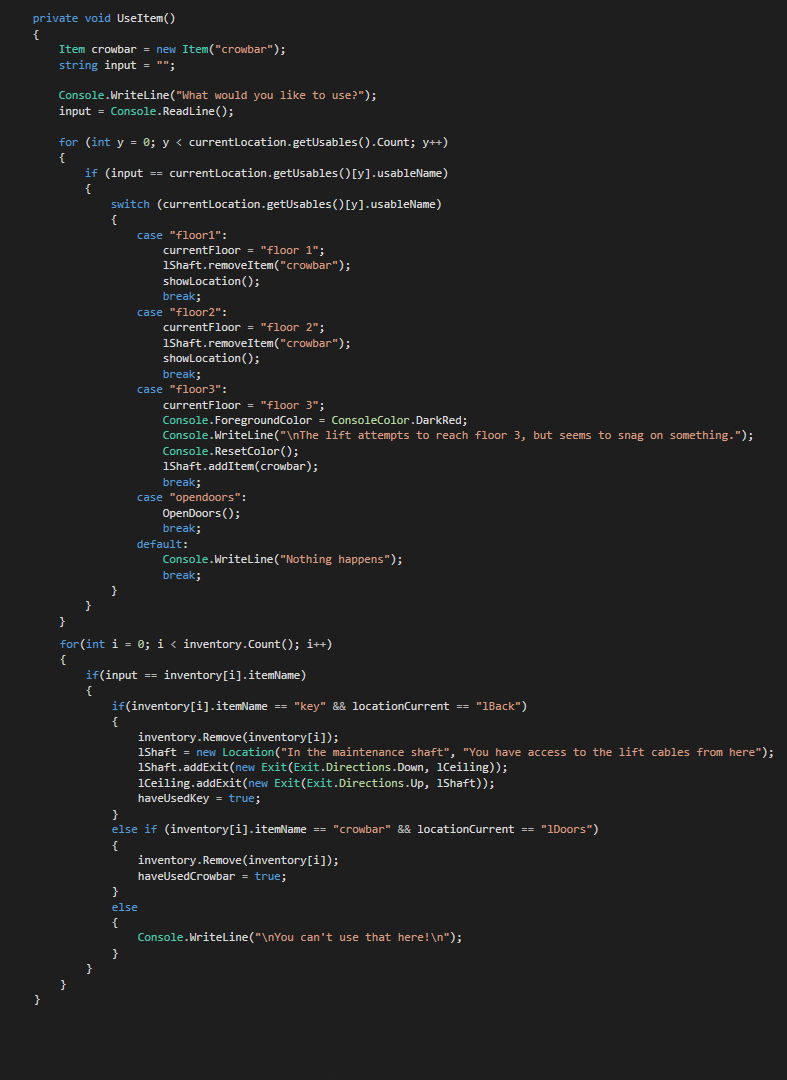
I also have a class called “Usables”, which does a similar thing but in reference to items in the environment e.g. a button on the wall.

# Inventory

My inventory system makes use of the “inventory” variables that were implemented in the source code to hold the items present in both the environment and your person. The code to handle it is located in two voids:



This is the code for taking an item. It uses a basic if/else statement to check if there are any objects in the current location for the player to take, then assigns the player’s input to a new variable. This variable is then checked against names of items in the current location (handled in Location.cs using the takeItem function in the source code). If the name matches, the item is added to the player’s inventory.



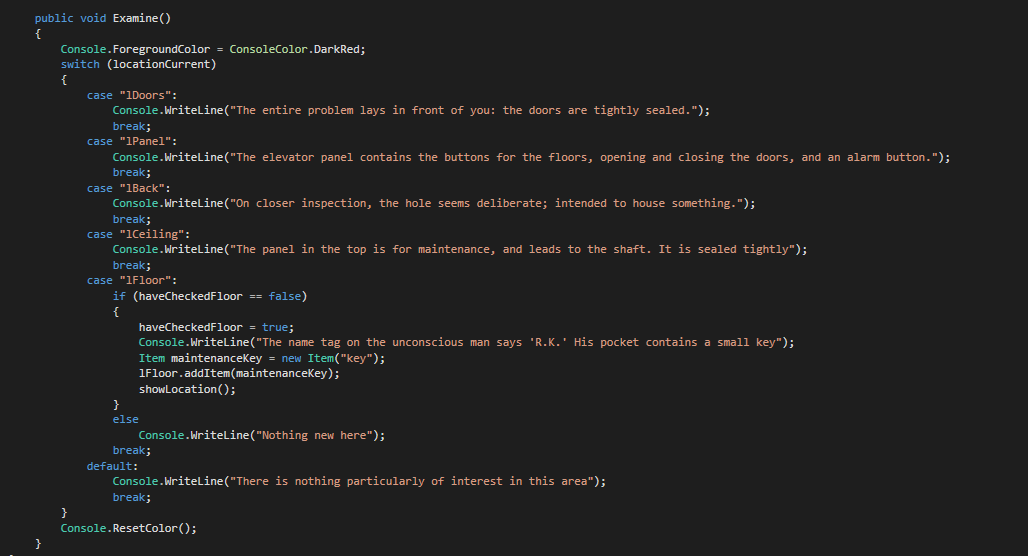
The use item method allows the player to use items from their inventory, but also allows them to use some objects in the environment e.g. the lift panel buttons. Both functions are wrapped in for loops, which check through all possible items in the environment/your inventory to see if your input matches any of them.

The code for the environment objects (“usables”) uses a switch statement, which executes a specific code block whenever a particular object is used. For example, using “floor3” takes the user to the third floor, gives them some story info, and adds a crowbar to the maintenance shaft. I felt this was apt because there are only a few environment objects in the entire game, and even fewer that actually did things when you pressed them. Thus a switch statement made sense for executing specific code related to each object, as the low amount of objects that needed to have code attached would prevent the statement from getting too long winded.

The use item code does something similar, but using if/else if/else statements. This is because I wanted the player to only be able to use items in particular areas, to prevent items being lost prematurely, and to prevent having to respawn items, which wouldn’t make sense in the universe I’m presenting to the player.

# Looking Around

The player looks around the map using a combination of two functions. Firstly the showLocation method that was already implemented: I use the descriptions of locations to give the player a basic idea of their surroundings. There is also the “examine” method which gives the player some greater detail, and occasionally reveals items for the player to pick up.



The examine function makes use of a static string called “locationCurrent”. It checks against your location and provides additional info based on where you are.

# Moving Around

Like the examine function above, moving around is based on a public static string called “locationCurrent”. When an input is made, the current location’s name is checked against that string, and changes the location based off of four separate functions. The showLocation function gives information to the player regarding where they can go by checking against instances of the Exit class present in that instance of the Location class.

# Inputs

The program handles all inputs using a method called “doAction”, which parses in a player input, and initiates the appropriate function if said input matches one with code attached. The player cannot use an input that does not match the program’s dictionary; attempting one will return a blank line, showing nothing has been done. Player’s must instead rely on an existing dictionary of commands, that is shown to the player upon typing “help”. This avoids ambiguity when dealing with player input, reducing the chance of the game breaking, or otherwise doing something that the player is not expecting.

Input in regards to the use of objects follow a set pattern. This pattern entails using the command “use” to trigger the UseItem function, at which point the player will be asked which item they want to use. Items can only be used in specific places, so an attempted use of an item outside of will give the player a message: “You can’t use that here!”. A similar method is used for taking an item from the environment.

# References

<http://www.yoda.arachsys.com/csharp/constructors.html>

<https://msdn.microsoft.com/en-GB/library/0f66670z.aspx>

<https://www.codeproject.com/articles/18734/method-overriding-in-c>

<http://stackoverflow.com/questions/4718725/best-practice-to-avoid-invalidoperationexception-collection-was-modified>