Titan Rover Project

Program design

The program uses dictionaries to store both it's commands and variables for its commands.
The methods (commands) are named as a single unique character that matches they key used by the user to utilise them.

The dictionaries use the method names as their keys, and stores the methods with them.

The dictionary keys are displayed on the console informing the user which keys they can use.

How to use the program

To move the rover press the keys to build a command string and return.

The rover will check valid keys are used, and that the command will not take it out of bounds.

If invalid keys, or a command putting the rover out of bounds is given it will ask for new commands.

When asking for new commands it will provide information on the first letter in the command that was invalid or which letter of the command would of put it out of bounds with the coordinates that the out of bounds commands would have put it in had it been implemented.

If the command is valid it will execute it return the new coordinants and ask for its next command.

Adding new program functionality

Communicating with the user

To improve communication with the user the Directions class methods could contain a string of information on their method. For instance "North" for N. The dictionary could store the MethodInfo and the string in a new struct for storing this data together. Or the string could be accessed through MethodInfo.

Commands

The dictionaries can be added to for more movement and orientation functionality. Examples are in the code. Simply uncomment the example to see the functionality added.
The code only needs changing in one place as the dictionaries themselves are used for checking input and communicating with the user.

The program takes a string not a string array. Therefore all input commands must be a single character and so must the string dictionary keys. Therefore when adding a movement to the moveAction dictionary it must be given a unique single character key and when adding an orientation method to the Directions class (used to construct the orientationCommands dictionary) it must alse be a single unique character.

Coordinates

The Coordinates class can be altered to be created with different bounds for its coordinate system. To do this make the rover constructor take the values desired for the coordinate class constructor and have it supply them when it makes its coordinates. The Coordinates class already has defaults these could be copied into the rover constructor.

The coordinate system does not use 2D array or lists and does not store information. This functionality may be desireable in future with the max and min defining the array size.

To create functionality allowing you to alter the coordinate system after the rover is active use the getter and setters. Within the getters and setters build in code to protect against shrinking the coordinate bounds such that the rover is out of bounds and or to translate the rovers current coordinates to match the new system being used. For example if grid location 3,3 became 4,4 because the grid was extend West and North one square, with 1,1 being the most North Westerly square.

If using coordinates for mapping with different maps with different bounds on the same rover it is recommended that the rover does not have pocession of the delegate but instead the coordinate systems or another new class encapsulating it does.

Design Decisions

The brief asked for a string not a string array to be used. An array would allow for "NW" to be a command. An array could be implemented in future by requring the command to come in a format for example N,M,NW,M where array elements occur between to comments. The project therefore to go "NW" would need a single character to represent this for example X. (This is not very descriptive therefore in the adding functionality sections it is suggested the commands have a describing string for the console to use).

A delegate has been used for storing the orientation methods. However a multidelegate could have been used.

The multidelegate would allow the North-West "NW" command without making a new specific method for it. NWM would result in North-West movement because it would move the rover both North and West. Whereas the current program uses an ordinary delegate so only stores the last orientation given to it, therefore NWM would execute a command equivalent to WM.

Note that the multidelegate would need to be emptied after the last M in a string before a new orientation, for example where the astrix occurs in NWMM*NM. This makes a clear command desireable which was not in the brief. The code could automatically clear the multidelegate but consider that as the default orientation is South, to go East the user would have to clear the South by either cancelling it with a North command NEM or clearing it (lets say clear is C) CEM otherwise EM would result in a South East movement.

This multidelegate approach was also not used because orientation and movement commands would have overlapping roles as NNNNM would result in the rover moving north four times.

However a benefit of this approach would be if using vectors the rover could diagonally go to a position on its grid, therefore go the most effeicient route but still always end up in an exact initiger grid location.