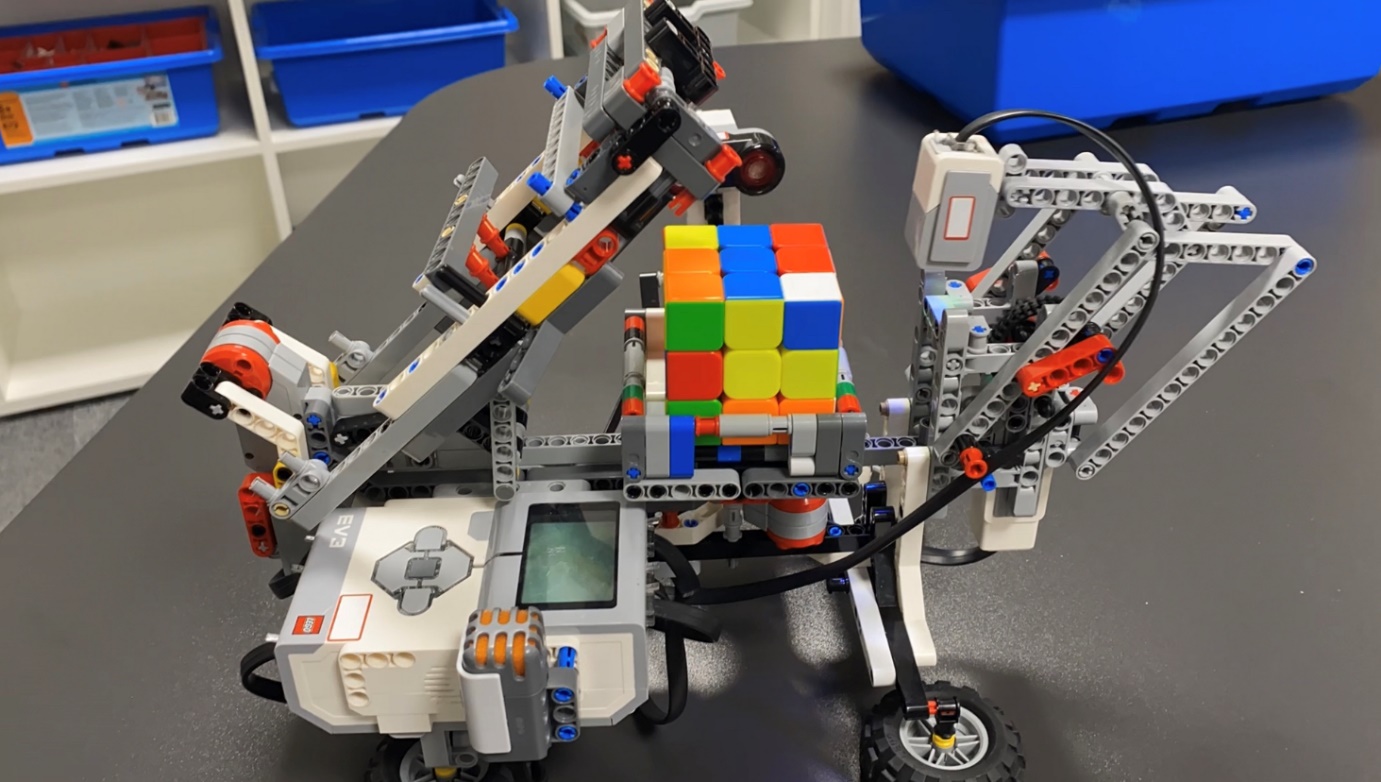
**Introduction**In this report, I will give an overview of Lego Mindstorms and some of the key principles required in programming robots, as well as an in-depth explanation of how the robot we developed works. In addition, I will review the work we achieved together as a team and reflect upon it accordingly.

**Robot Description**Our group had decided that we wanted our robot to solve a Rubik’s cube. Initially, we decided to do some research on this project in order to find out whether anyone else had attempted it before and if we could gather information that would help us design the robot. These instructions - http://mindcuber.com/mindcub3r/MindCub3r-Ed-v1p1.pdf - we came across were perfect for creating the robot we desired. The cube sat on a rotating platform (which was achieved by having it attached to a motor), while it was flipped by an arm attached to another separate motor. After lowering the arm onto the cube, the platform turned 90 degrees, turning one of the sides.

Several factors made building the robot challenging. Our supplies were limited and we lacked many parts, which we searched for in boxes of spare parts. This took up much more time than we had anticipated so in the end, multiple pieces had to be substituted. As an example, rather than using one long piece, we used two shorter cross-head sticks and a Lego connecting piece to ensure the two stay together. We also faced a problem with the platform being too big for the cube with which we tested the robot. We tried several different solutions, including: creating thin attachments inside to make it slightly smaller and using a different larger cube. Neither of these solutions worked, and in the end we left the platform as it was with no other alterations. We have yet another motor attached to a colour sensor that scans every tile on the cube and records its colour for future use. Near the end of the process, we decided to replace this with a button that sits in the same place but is easier to use.



**Key Concepts**

**Lego Mindstorms**A Lego Mindstorm is a small brick that is able to run Java Virtual machines on it. Each Lego Mindstorm has buttons that can be used for input, a speaker and a display to show the interface and execute outputs. Java programs can be uploaded and run on these devices. They can support up to 4 motors and 4 sensors at a time (8 total ports) and each can be used in a unique way to observe surroundings and gather input.

**Sensors**Sensors are devices that measure things in their surrounding and send their results back to the brick. Sound sensors measure how loud a noise is; ultrasonic sensors release a beam and measures the duration it takes to return – this gives an accurate approximation of how far away any objects are. There are also buttons which have three states – not pressed, pressed, and held down – and a colour sensor which emits light onto the surface, and determines the colour based on what is absorbed by the object and not returned.  
Our robot uses 2 of these sensors – the colour sensor and a button.

**MovePilot**The MovePilot class is useful and simple in Lejos programming; it enables the movement of the robot. It is able to make complex and exact movements once it knows the diameter of the wheels, how far they are from the robot’s centre, and what chassis the robot uses. (The chassis serves as a control system in order to drive the robot).  
This class can prevent the robot from accidentally turning to one side - a common problem - when it starts or stops moving, where this would take multiple lines of adjustment to fix without it. It can also set a maximum speed and acceleration.

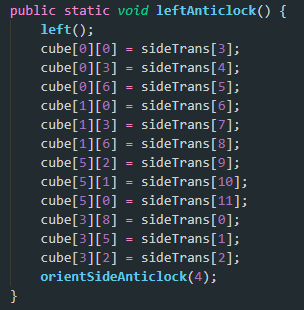
**Navigation**Due to MovePilot's accuracy, a class called OdometryPoseProvider lets you track the robot's location at any time. Robot odometry is the process of determining where the robot has moved by using information from sensors or motors. The class measures its position and calculates where it is, making it possible for the robot to navigate complex environments while also being able to move to set waypoints when given only starting coordinates or to avoid obstacles in the way of it’s path.

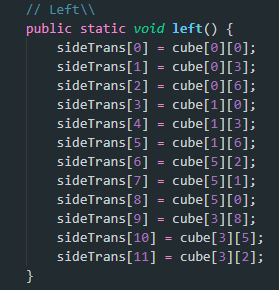
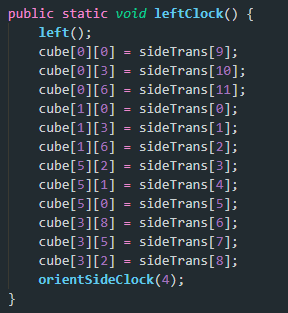
**Behaviour**Lejos supports arbitrators and behaviours. A behaviour is an action that only runs if specified conditions are met. For example, our robot will only start solving the cube once you press the button attached to the side. This is to prevent it from starting before the cube has been placed on the platform.  
Arbitrators are lists of these behaviours, and they are written in order of priority – going from lowest to highest. Each behaviour is checked regularly to see if the conditions have been met, and if they have then it will be executed immediately. In the event that two or more behaviour’s conditions are met, the one with the highest priority runs first.

**Technical Content**Classes

**CubeSolver2**Firstly, there are 3 arrays of Strings initialised: sideTrans (which shows the translation of the cube), orientTrans (which contains the orientation) and cube (which is a nested array of each side of the solved cube). Then each motor is assigned to a variable, mTwist,, armMove and mFlip. The colour sensor is also initialised to a variable called sample.

Then there are many private final variables that initialise constants showing how much each arm moves and what motors will need a slight correction through corrective movements of each motor.This is also where all of the movements for the cube are defined. Each one is a method - called up, down, left, right, front and back – with two arguments: int clockwise and int turnTimes.   
Up flips the cube so that faces upward, down flips the cube so it faces downwards and the others flip the cube so the named side is facing that specific way (left faces left etc).

There are also methods called x(), xClock() and xAnticlock() where x is the name of each face (up, down, left, right, front, back). In each one, each index of sideTrans is set to equal the correct position of the cube.   
xClock() translates the cube face by 90 degrees clockwise while xAnticlock translates the cube by 90 degrees anticlockwise. There are in total 3 methods for each side like this, making a total of 18 methods. Examples for all three of these methods are below.



Shuffle() is what creates the initial shuffle of the cube and uses the random function to achieve this. Depending on what random number is chosen, it adds a new character to the string to form a complete shuffle and does this 12 times. Once the shuffle is created, repetitionCheck is run to make sure that if – for example – there are two “r” characters next to each other “r2” is returned instead.

notationToMovement takes a String which then runs specific moves (eg xclock() that was explained earlier) dependent on each character of the string. Once the entire string has been run through, we now have the moves to solve the cube and then the moves are executed via robotMovement which runs the methods x(int clockwise, int turnTimes) where x is the side to move.

cornerOrientationCheck ensures that the cube is facing the correct way through a switch case that checks each colour and if its in the wrong orientation, provides a string to notationToMovement to correct this.

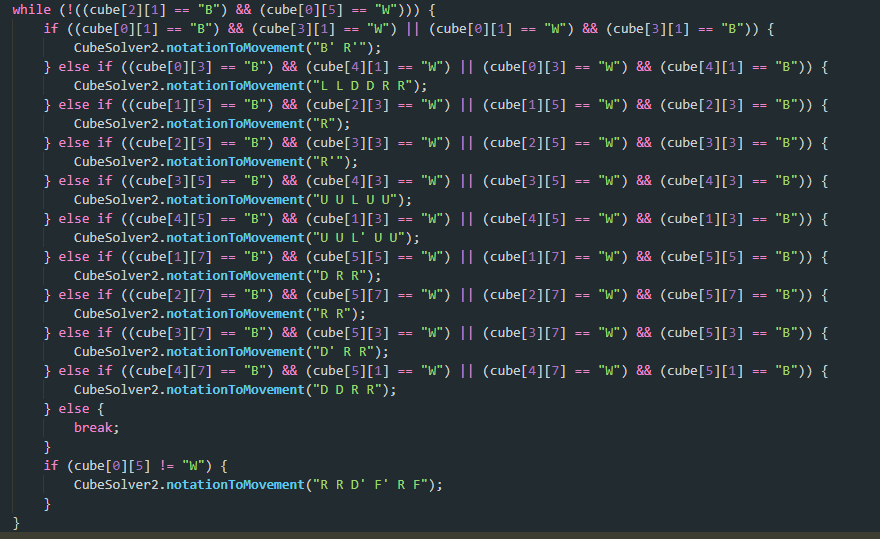
**EmergencyStop**Three BaseRegulatedMotor variables are created and these are set to the 3 arguments given in the EmergencyStop constructor. When the left button has been pressed, takeControl() returns true and action() is then run. mTwist, armMove and mFlip (the three arguments passed in the constructor) are stopped and the system is exited. There is nothing in supress() as the system will be exited.

**LowBattery**takeControl returns true when the battery level goes below 5 volts, and “Low battery, shutting down” is displayed on the screen for 1 second before the system is shutdown again with Main.end(). There is no code in supress() as the system has been shut down.

**Menu**Menu creates a Boolean runMenu which is set to true initially, a SampleProvider sp and a float array Samples. A constructor is made taking a SampleProvider argument. When the sample input provides 1, the button has been pressed and takeControl returns true. action() is then run which shows each group members name for 1 second, then the screen is cleared and outputs the code version. The screen is cleared once again and displays “Press enter to run” and the code then waits for enter to be pressed and released. Once this happens Main.runRun is set to true and runMenu is set to false. This means that the Main class will then proceed to run the Run class and the menu Behaviour is stopped. Supress() contains “runMenu = false;”

**Main**Main creates 4 behaviours – emergencyStop, menu, lowBattery and run; each one of these behaviours being a new instance of the corresponding behaviour classes. A Boolean runRun is created so that we know when to run the Run behaviour. An arbitrator is then created with the behaviours run, menu, lowBattery and emergencyStop in that order of increasing importance. A contructor for the arbitrator is also created for use in the main method. A method called end() is created to stop all motors, stop the arbitrator and exit the system. The main menu is used to start the arbitrator and stop it after.

**Run**Run is a behaviour and executes when Main.runRun returns true. While run is true, then the cube is read, and depending on what colours are where, a certain while loop is executed with if statements and a certain string is set to execute within CubeSolver2 using notationToMovement in order to turn the cube. Here is an example



**Reflection**I believe that our group worked well, however we could have done better. Firstly, due to unavoidable circumstances, I was not on campus for multiple weeks and had to work remotely where I could not test code properly with the robot, and had limited contact with everyone. I do believe that I made up for this by doing a large amount of building the robot near the beginning of the project and also doing a lot of the programming and cleaning up of the code in the last few days. These were not my only contributions – I actually found the initial design and instructions for the robot – however they are my most noticeable. I also believe that Member3 did the majority of the coding, as he had coded a Rubik’s cube solver for a previous project and knew how to do this with ease, whereas me and Member1 did the colour sensor as well as the arbitrator and behaviours. For the last and most important few days before the deadline, unfortunately we could not contact Member2 which meant me and Member1 did a lot more work however Member2 did contribute to helping test the code earlier on in the project.  
The hardest part about writing the program was making sure that all the behaviours worked as intended. Unfortunately, this was a lot more difficult than the group anticipated and I spent the majority of 2 days sat in the lab testing this. I was unable to easily get the EmergencyStop behaviour to work as even when I pressed the button and the condition was met, the code would not stop running. If I could do this project again, I would code and implement the behaviours as I write the code, and not try to split the code up at the end. This created a lot of issues and was unnecessary to do later on.  
Our robot did end up correctly solving a Rubik’s cube and shuffling it, however, every now and again the arm would fail to flip the cube correctly and the algorithm would continue but not with the correct orientation. This meant that the cube would not be solved at the end. In order to fix this, we could have had the colour sensor scan the cube after a certain amount of moves to ensure that it was still in the correct orientation – otherwise it would restart the solve.

**Conclusion**Overall, I believe that our robot was successful as we managed to program it to solve a cube and had it working consistently enough to solve an 80 move algorithm. In addition, our team worked well and communicated to each other when needed and also helped each other if we didn’t understand or got stuck with a certain idea.