You've already learned a lot about programming in the first five lessons, but there's also been a hand-waving of concepts by saying "ignore this for now, we'll learn about it later". This has enabled you to spend more time focusing on writing code and playing with the robot. However, it can also feel a little overwhelming when you're looking at a lot code that you don't fully understand. Similar to what we did in lesson four with methods, this lesson is going to focus on helping you build a deeper understanding of object-oriented programming and what classes and objects are. These are some of the fundamentally most important concepts in Java and many other programming languages, so take your time with this lesson, focus, repeat the reading and/or videos as necessary, and don't worry if it seems overwhelming at first. If you go through the material and projects carefully, you *will* come to understand it, and that will help you understand what's going on in the robot code much better as well. For the hands-on portion of this project, first you'll follow along with a video, and then you'll modify the Romi code to use two of the built-in LEDs – red and green – to indicate whether or not the Romi has been rotated from its original orientation (for example, if you were to drive it and it started to veer off course instead of driving straight ahead.)

Start out by watching this video and following along in a blank non-robot project in VSCode. If you don't remember how to set up a non-robot project in VSCode, you can refer back to lesson 4. This video gives a simple explanation of objects, how they have methods and variables, and how you can use them: https://www.youtube.com/watch?v=0NPR8GFHNmE

After you watch the video and code along with it, read this page, from the start through section 2.9. Don't worry about anything that comes after section 2.9: https://www3.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html

--

Now that you've made your way through these materials, you should have a decent idea of what object-oriented programming (OOP) is, and robot code will probably make a little more sense to you. There are still a lot of concepts to cover in part two, but you've done enough reading for one lesson so let's get our hands dirty with some Romi code.

Measuring Orientation: Gyroscopes

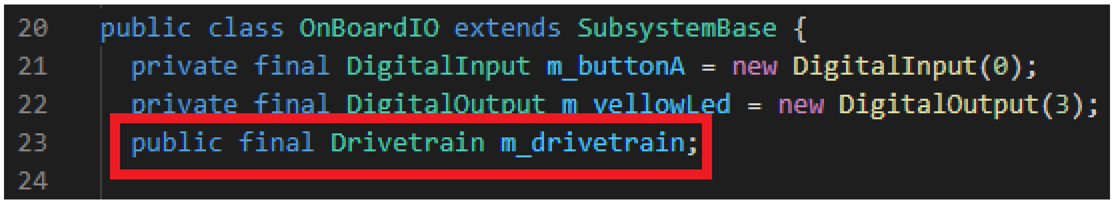
The Romi has a built-in gyroscope, or *gyro*, which is a device that measures changes in orientation of the robot and can also add up all the changes over time to measure the current orientation at any given moment. Having access to this data opens up a lot of possibilities when it comes to robot code. One example of this is the autonomous program you saw run in the very first lesson, where the Romi turns around twice. How does it know how much to turn, to turn around without turning too far (or not far enough)? While this could be achieved by measuring exactly how far each wheel spins, the most reliable way to do this is to use the gyro, because, for example, the wheel could slip. Another great use of a gyro is detecting whether or not your robot is driving straight, because there are a lot of ways your robot could drive a bit crooked. One motor might be slightly more powerful than the other. One side of the drivetrain might have more mechanical resistance than the other. The surface you're driving on might be a little bit uneven. Those are just a few examples; there are a lot of things that can go wrong, but with a gyro, you can detect all these things, and correct for them. In a future lesson we'll correct for them, but today we'll just worry about detecting them. We'll write a program that uses the built-in lights to light up either red or green, based on whether or not the Romi is still facing the same direction it was when the program started running.

Accessing Class Data

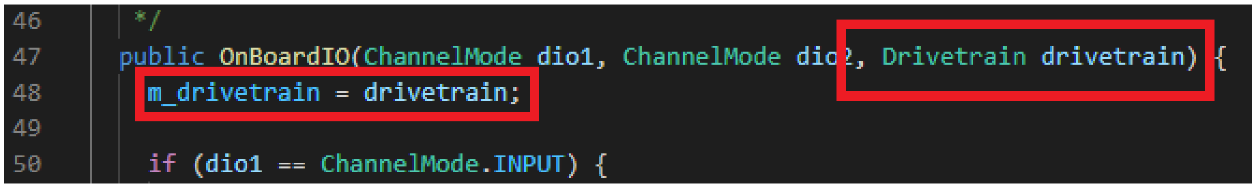
To do this project, we're going to need to access a few variables that, by default, we can't access. We'll talk more about this in the second part of the object-oriented programming deep dive, next lesson. There are reasons why these variables should not be accessible to us, but for this project, we're going to make them accessible anyway. Follow these steps to get everything setup for this project:

1. Create a new RomiReference project like you've done a couple times already.
2. In RobotContainer.java on line 31, there are two instances of the word "INPUT". Change these both to "OUTPUT". This tells the Romi to use the red and green LED lights as output.
3. In Drivetrain.java on line 32, change the first word of the line, "private", to "public".

With that out of the way, it's time to get started. This program will utilize your newly gained knowledge about classes by accessing variables belonging to various objects in the program. In this project, we're going to create an interaction between two subsystems: the drivetrain, which is where the gyro exists, and the on-board IO, which is where the lights exist. Go to OnBoardIO.java and create a class field for a Drivetrain object. We'll populate this field with the Drivetrain object that is declared in RobotContainer, so that the OnBoardIO object can access the gyro that belongs to RobotContainer's Drivetrain object. After declaring the Drivetrain object, initialize the OnBoardIO's *constructor* method by adding a parameter of type Drivetrain, and setting the class's Drivetrain object equal to the Drivetrain object that is passed into the constructor. Remember, a constructor is a method in a class that exactly matches the class's name and has no return type. So in OnBoardIO, the constructor is the method declared on line 46 by the text "public OnBoardIO…" (it will be on line 47 after you declare the Drivetrain object. Here's an example of the declaration and initializations:

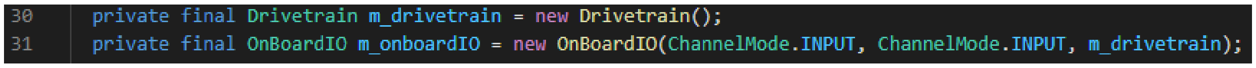


```
20  public class OnBoardIO extends SubsystemBase {
21    private final DigitalInput m_buttonA = new DigitalInput(0);
22    private final DigitalOutput m_yellowLed = new DigitalOutput(3);
23    public final Drivetrain m_drivetrain;
24
```



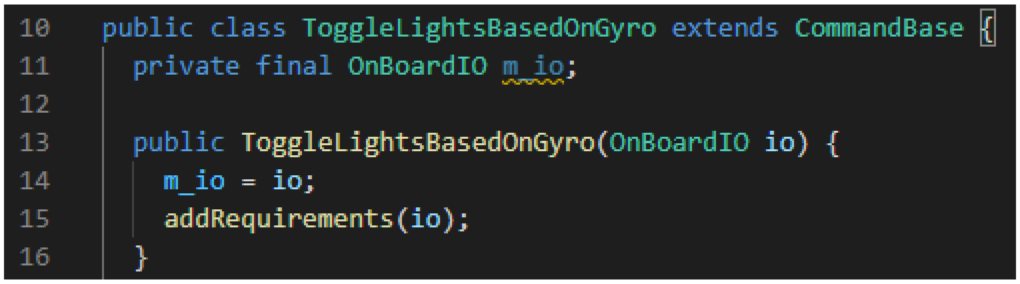
```
46     */
47    public OnBoardIO(ChannelMode dio1, ChannelMode dio2, Drivetrain drivetrain) {
48      m_drivetrain = drivetrain;
49
50      if (dio1 == ChannelMode.INPUT) {
```

After doing this and saving the file, you'll have an error in RobotContainer.java because the line that initializes the OnBoardIO object there doesn't send a Drivetrain object. Navigate to line 31 in RobotContainer.java and update it to pass in the Drivetrain object that is created on the prior line.



```
30    private final Drivetrain m_drivetrain = new Drivetrain();
31    private final OnBoardIO m_onboardIO = new OnBoardIO(ChannelMode.INPUT, ChannelMode.INPUT, m_drivetrain);
```

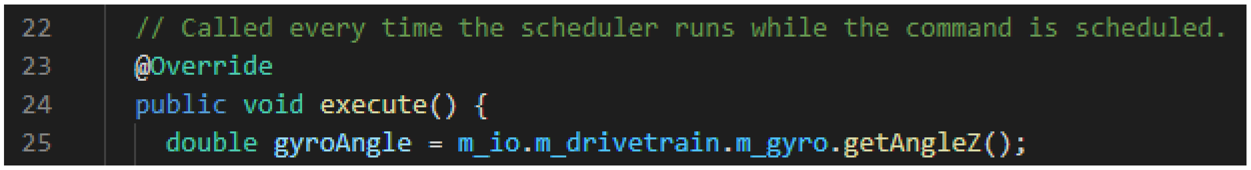
You're now done with the plumbing and ready to create the command that will actually *do* things. Create a new command in the commands folder called ToggleLightsBasedOnGyro. After creating the command, add a class field of type OnBoardIO called m_io. You can declare this one using the keywords "private final", as shown in the image below. After creating the field, initialize it in the constructor, also shown. The last thing you'll need to do is call the "addRequirements" method and pass in the OnBoardIO that is sent to the constructor. This is the line that tells the command which subsystems it needs to use. This is how the program manages which commands are running on a subsystem because any given subsystem can only run one command at once. You can run multiple commands at the same time if they are on different subsystems, and the addRequirements method specifies which subsystems each command uses. Although this command that we're making will *read data* from the drivetrain subsystem, it only *acts* using the OnBoardIO subsystem, where it will turn lights on and off. So we do not "require" the drivetrain subsystem for this command in a programming sense, even though in an English sense it does need a drivetrain subsystem to read data from. Here's a completed example:



```
10  public class ToggleLightsBasedOnGyro extends CommandBase {
11    private final OnBoardIO m_io;
12
13    public ToggleLightsBasedOnGyro(OnBoardIO io) {
14      m_io = io;
15      addRequirements(io);
16    }
```

Now it's time to add the logic that reads the angle from the gyro and then turns the lights on and off accordingly. Doing this will be an example of how objects form a hierarchy of data. The command you're working on is literally a class – you can see this on line 10 where you declare "public class…". This class has an OnBoardIO object, m_io. Every OnBoardIO object has a Drivetrain object because you gave it one on line 23 of OnBoardIO.java in a previous step. Every Drivetrain object has a RomiGyro object because of line 32 in Drivetrain.java (one of the lines you modified while setting up for this project.) So from your ToggleLightsBasedOnGyro class, you can chain through your data hierarchy to get to the gyro: ToggleLightsBasedOnGryo -> OnBoardIO -> Drivetrain -> RomiGyro. The RomiGyro class has a method called "getAngleZ()", which returns a double value that represents an angle in degrees (you can see this if you open RomiGyro.java in the "sensors" folder, and look on line 108.) As you learned in the materials for this lesson, you can use dots (periods) to chain through your objects. So to call the getAngleZ() method, you can use your class variable m_io, and do "m_io.m_drivetrain.m_gyro.getAngleZ()".

While a command is running, its *execute* method is called continuously. We want our lights to update continuously, so this is where we'll put our code. Navigate to the execute() method in your command (approximately line 24), and as the first line of the method, declare a variable and set it equal to the output from the getAngleZ() method call described above. Here's an example:



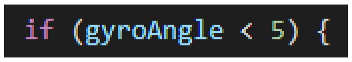
```
22    // Called every time the scheduler runs while the command is scheduled.
23    @Override
24    public void execute() {
25      double gyroAngle = m_io.m_drivetrain.m_gyro.getAngleZ();
```

Now you have a variable that contains the current orientation of the robot, relative to the robot's orientation when the program started running, which will be whenever you press F5 and the code loads onto the robot. Since execute() is called continuously while the command is running, this variable will continue to be updated during execution.

Now that you have the angle, it's time to turn lights on and off. To do this, you'll want to use two methods belonging to the OnBoardIO class – setGreenLed(boolean) and setRedLed(boolean). Notice how those methods are written here in plaintext, with the word "boolean" in the parentheses. This demonstrates that in order to use them, you need to send a boolean variable. This makes sense – lights have two states, on or off, so you can pass either true or false to the methods, to turn the light on or off. There is one method for the green light and one method for the red light. In this project, we'll set the green light to on if we're on course, and the red light to on if we're off course. In both cases we'll set the other light to off, so there will only ever be one light on at a time.

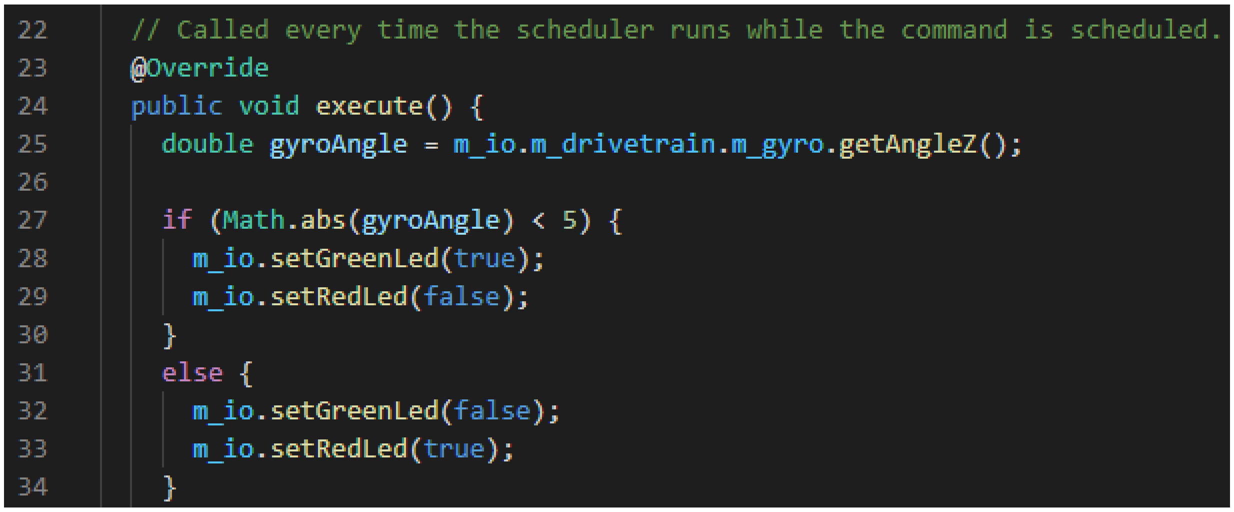
You might want to turn the lights to one configuration, and you might want to turn them to a different configuration, depending on the gyro angle. That means you're going to need a conditional statement. "Conditional" is just a fancy word for an if statement. If the gyro angle is zero, then the robot is facing straight ahead, and if it's not, then the robot has turned a little bit. Of course, in practice, the gyro is sensitive, and the angle is not going to be *exactly* zero, so you're going to need to check if it's in some range. If you make the range too big, say, 90 degrees, then you could veer far off course before detecting it. If you make the range too small, say, one quarter degree, then you're going to detect that you're off course even when you're mostly straight. Finding the perfect range depends on a lot of things, but for this project, a range of +/- five degrees works pretty well.

You could measure if the gyro angle is less than 5 degrees pretty easily, with a line like this:



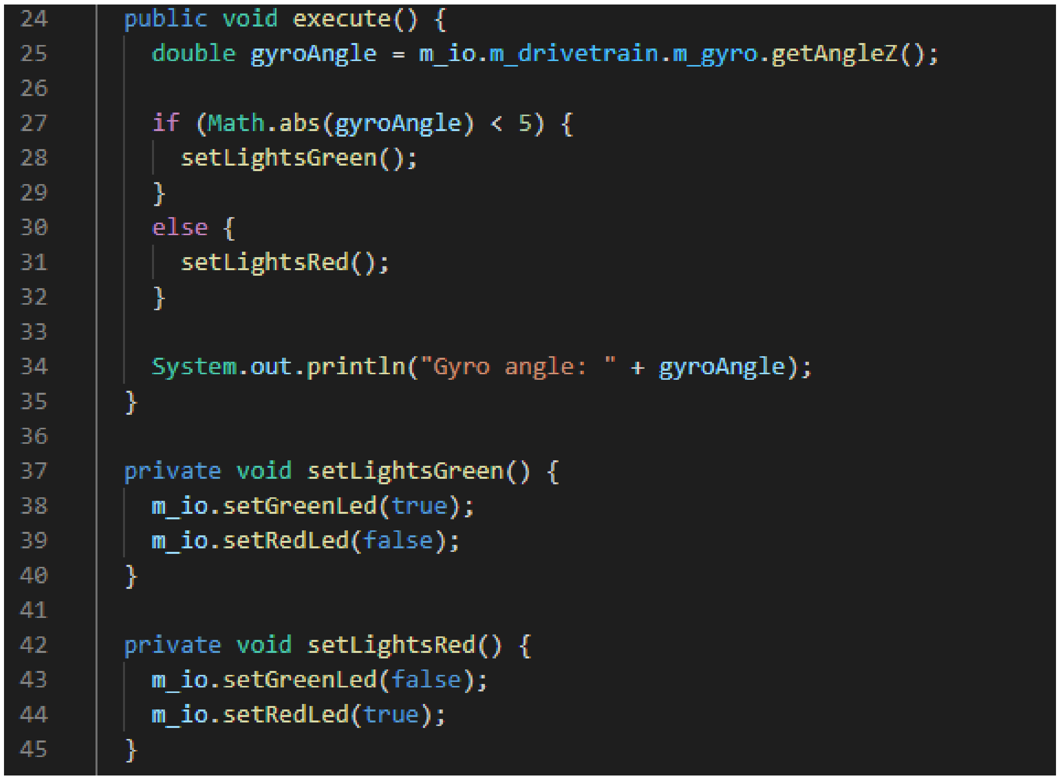
```
if (gyroAngle < 5) {
```

However, the robot can turn in two different directions. If you turn the robot to the right, the gyro angle will increase. If you turn to the left, it will *decrease*. This means that if you turn 90 degrees to the left relative to the starting point, the gyro angle would return -90. This is indeed less than 5 degrees, so the if statement above would return true, even though you'd be very far from straight. Thankfully, there's an easy solution to this, using something called "absolute value". The absolute value of a number is simply that number, but *not* negative. For example, the absolute value of 10 is 10. The absolute value of -10 is also 10. So the only difference is that if the number is negative, it becomes positive. If it's already positive, there's no change. Java has a built-in method to return the absolute value of any number: Math.abs(double). Math.abs(gyroAngle) will return whatever the gyro angle is, but positive. You can use this function in your if statement to check if the absolute value of the gyro angle is less than five. Write this if statement, and if it's true, set the green LED to true and the red one to false, and vice versa if the if statement is false. Here's an example:



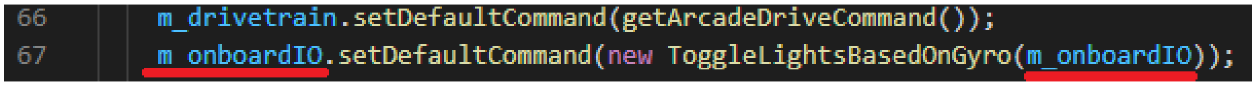
```java
22    // Called every time the scheduler runs while the command is scheduled.
23    @Override
24    public void execute() {
25      double gyroAngle = m_io.m_drivetrain.m_gyro.getAngleZ();
26
27      if (Math.abs(gyroAngle) < 5) {
28        m_io.setGreenLed(true);
29        m_io.setRedLed(false);
30      }
31      else {
32        m_io.setGreenLed(false);
33        m_io.setRedLed(true);
34      }
```

We could be done with this command at this point, but let's make a couple quick upgrades before we finish. Since we're still practicing our mastery of using methods, let's split out lines 28 and 29, and 32 and 33 above, into their own methods. Beneath the execute() method, create two more methods called "setLightsGreen()" and "setLightsRed()". You can declare both of these with "private void". Move the lines of code identified above into these methods, and inside your if statement in the execute method, call your newly created methods. Lastly, you may want to add a SOP line that outputs the gyro angle, so you can see it in real time in the terminal. This isn't strictly necessary because you can also see it on the robot simulation interface, but you can add it if you want. Here's an example of the command with all these steps completed:



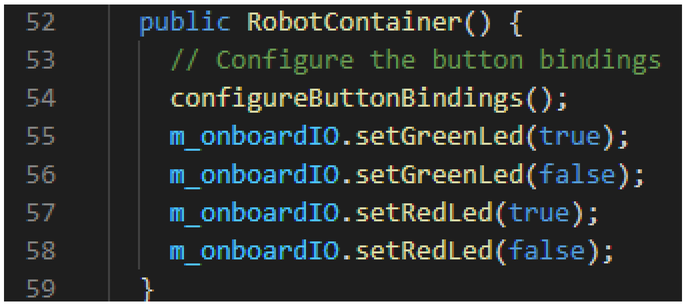
```java
public void execute() {
  double gyroAngle = m_io.m_drivetrain.m_gyro.getAngleZ();

  if (Math.abs(gyroAngle) < 5) {
    setLightsGreen();
  }
  else {
    setLightsRed();
  }

  System.out.println("Gyro angle: " + gyroAngle);
}

private void setLightsGreen() {
  m_io.setGreenLed(true);
  m_io.setRedLed(false);
}

private void setLightsRed() {
  m_io.setGreenLed(false);
  m_io.setRedLed(true);
}
```

With that done, your command is completed. The only thing left to do is set your command to be the default command for the OnBoardIO subsystem. Since no other commands are issued to the OnBoardIO subsystem right now, setting your command as the default command means that it will run all the time, and therefore your execute method will run all the time. Navigate to the end of line 65 in RobotContainer.java and press enter. Then use the setDefaultCommand method you see on line 65, except call it for your OnBoardIO object instead of the drivetrain (remember, your OnBoardIO object, which is declared on line 31 of this file, is called m_onboardIO.) On line 65, the value passed to setDefaultCommand is a method call, but for the the line you're adding, you can just use the *new* keyword to create an instance of your command. Remember, commands are classes, so you create instances of them. To create an instance of your class, you need to send in the values its constructor requires. In this case, it requires an OnBoardIO object, because the OnBoardIO object provides access to everything your command needs – the lights to toggle and the drivetrain object through which to find the gyro angle. Conveniently, you only have one OnBoardIO object in your code, which makes sense because your robot physically only has a single on-board IO. The same m_onboardIO object that you're assigning the default command to, can be sent in to your ToggleLightsBasedOnGyro constructor call, like so:



```
66      m_drivetrain.setDefaultCommand(getArcadeDriveCommand());
67      m_onboardIO.setDefaultCommand(new ToggleLightsBasedOnGyro(m_onboardIO));
```

Now there's just one tiny step to go. Navigate to the end of line 54 in RobotContainer.java, press enter, and then add four lines like so:



```
52    public RobotContainer() {
53      // Configure the button bindings
54      configureButtonBindings();
55      m_onboardIO.setGreenLed(true);
56      m_onboardIO.setGreenLed(false);
57      m_onboardIO.setRedLed(true);
58      m_onboardIO.setRedLed(false);
59    }
```

All this does is initialize the values so that the lights turn on as soon as your code starts. With that, you're done – deploy your code! Make sure not to rotate your robot by hand before you enable it. When you enable teleoperated mode, you'll see the green light illuminate. If you start rotating your robot, either by driving it or by hand, you'll see the lights change as the angle of the robot moves away from its initial orientation and change back if you move the robot back.

In this lesson, you learned about OOP, and saw an example of how you can use classes and objects in code to logically interact with different parts of your program through a class hierarchy. You also saw an example of how you can integrate different subsystems on the Romi with each other to create some cool functionality. This was a longer lesson with a lot of important concepts, so don't worry if you feel like you need to review any part of it. Although there are still a couple deep dives left to go before you'll understand everything you see in the code, you're making great progress. We'll continue to build on these skills in future lessons.