

In lesson 6, you learned a lot about object-oriented programming, but there's still more to learn, so we're going to focus on that in this lesson. For the project, we'll combine the new material you learn with review, by revisiting the project from lesson 6 and doing it in an easier and better way using the new knowledge you have.

You previously read this page through section 2.9:

[https://www3.ntu.edu.sg/home/ehchua/programming/java/J3a\\_OOPBasics.html#zz-2.10](https://www3.ntu.edu.sg/home/ehchua/programming/java/J3a_OOPBasics.html#zz-2.10) Today, continue doing so, starting at section 2.10 and reading through section 2.17. Stop when you get to section 3. After you do, you'll have a much better understanding of words you've been seeing pop up everywhere in the robot code, specifically "public", "private", and "final".

--

Having finished the reading, you're now ready to dive into the project, which is a redo of the project from lesson 6 but applying the principles you learned from today's reading. Create a new RomiReference project called GyroLights 2, and just like last time, change the "INPUT" words in RobotContainer line 31 to "OUTPUT".

At this point in lesson 6, the next step was to change the RomiGyro variable declared on line 32 of Drivetrain.java from private to public. However, this time around we do NOT want to do that. As a general rule, it's better to have as many variables be private as possible. This is a safety measure so that external sources do not make unwanted changes to variables that could break your program. That might seem silly – you're the author of your program, so as long as you don't do that, what could go wrong? Well, in theory, you could get away with making everything public and never run into issues. However, in practice, most software projects include many people, not just one person, and no individual understands every line of the program. Team projects can easily be millions of lines. Somebody might access a public variable and change it without realizing how it affects something else. Even if you're not working on a team project, you're a human and humans make mistakes. It's surprisingly easy to forget all the details of your own program, especially after some time passes. If you're working on a tiny project that takes only a couple of hours to finish then pragmatically, yes, it's probably fine to make everything public. But projects have a way of growing past original expectations, and it doesn't take much growth before making everything public becomes a problem. Accordingly, it's a good idea to follow best practices and just make everything private if you can, so this time around we will not make the RomiGyro object public.

Following the same steps as last time, the next step was to create a public final Drivetrain object in OnBoardIO.java. We do need a Drivetrain object, but this time we don't want it to be public or final. As you learned in the reading, final means that the value will not be able to change once set, and we don't want that this time for reasons you'll see shortly. So this time, declare a private Drivetrain object:

```
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     private Drivetrain m_drivetrain;
24 }
```

Now it's time to practice something else you just learned about: overloading methods. To overload a method, you simply create two methods with the same name, but with different parameters.

In this case we're going to overload the constructor by leaving the original constructor and creating a new one that accepts a Drivetrain object. In lesson 6, we simply modified the original constructor. Create a new constructor with the same two ChannelMode parameters, plus a Drivetrain parameter. You'll need the functionality of the original constructor in addition to the new code you write, so the easiest thing to do is to copy and paste the old constructor and then add to it. In your constructor, initialize the OnBoardIO's drivetrain variable. Here's an example:

```
61 public OnBoardIO(ChannelMode dio1, ChannelMode dio2, Drivetrain drivetrain) {
62     m_drivetrain = drivetrain;
63
64     if (dio1 == ChannelMode.INPUT) {
65         m_buttonB = new DigitalInput(1);
66     } else {
67         m_greenLed = new DigitalOutput(1);
68     }
69
70     if (dio2 == ChannelMode.INPUT) {
71         m_buttonC = new DigitalInput(2);
72     } else {
73         m_redLed = new DigitalOutput(2);
74     }
75 }
```

In lesson 6, after we modified the constructor, it caused syntax errors in other files because the existing method calls no longer matched the parameters for the OnBoardIO (OBIO) constructor. By creating an overloaded constructor, we do not get those syntax errors, because the old code exists. This is a great feature when adding to large projects that have a lot of references to existing methods already. However, overloading is not always the right solution, and this project is actually a good example of that. We create an overloaded constructor that initializes the Drivetrain object. We will be using the Drivetrain object elsewhere in the method. But it is possible to create an OBIO object using the original constructor, which means it is possible to create an OBIO object that does not have an initialized Drivetrain object. If one were to do this, they would get an error if they later tried to use any of the code that relies on the Drivetrain object. That may or may not be a problem depending on the program. This is a tradeoff to consider whenever choosing whether or not to overload a method. Overloading can create better backwards compatibility and more flexibility, but you have to make sure that all of the options will work. In this case, if one used the old constructor and then tried to use the gyro lights functionality, it would most definitely not work, so if this were a production project, we would either not do the overload, or come up with some other workaround. Since it's not a production project, we'll do the overload and not worry about it too much, since the reason you're doing this project is to learn and thinking about the tradeoffs accomplishes that goal.

```

if (dio1 == ChannelMode.INPUT) {
    m_buttonB = new DigitalInput(1);
} else {
    m_greenLed = new DigitalOutput(1);
}

if (dio2 == ChannelMode.INPUT) {
    m_buttonC = new DigitalInput(2);
} else {
    m_redLed = new DigitalOutput(2);
}

```

If you look at the two constructors that now exist, you'll notice that the lines of code in the image above are repeated in both of them. A good rule is that whenever you see repeated lines of code, they should be broken out into a separate method. This reduces the amount of clutter in your code, and if you ever need to change those lines, you'll only need to change them in one place instead of tracking down all the places they're copied throughout your code. Let's practice that here. Split the lines of code picture above out into their own method called "initializeDios" (you can declare this one using "private void". Then call your new method in both of your constructors. Here's how the class looks so far:

```

47     public OnBoardIO(ChannelMode dio1, ChannelMode dio2) {
48         initializeDios(dio1, dio2);
49     }
50
51     public OnBoardIO(ChannelMode dio1, ChannelMode dio2, Drivetrain drivetrain) {
52         m_drivetrain = drivetrain;
53         initializeDios(dio1, dio2);
54     }
55
56     private void initializeDios(ChannelMode dio1, ChannelMode dio2) {
57         if (dio1 == ChannelMode.INPUT) {
58             m_buttonB = new DigitalInput(1);
59         } else {
60             m_greenLed = new DigitalOutput(1);
61         }
62
63         if (dio2 == ChannelMode.INPUT) {
64             m_buttonC = new DigitalInput(2);
65         } else {
66             m_redLed = new DigitalOutput(2);
67         }
68     }

```

Another thing you learned about from today's reading is getter and setter methods. Let's get some practice using them. You created a Drivetrain variable in this class, so create getter and setter methods for it. They'll look like the photo below. Note that the order in which you put

methods in a program does not make a functional difference, although it can make a difference in terms of how easy your code is for humans to read. Everyone seems to have their own style for how to order methods. The main reason you need to understand this right now is because your line numbers may or may not match exactly with the sample images if you're putting your methods in different places, and that's OK.

```
70     public void setDrivetrain(Drivetrain drivetrain) {
71         m_drivetrain = drivetrain;
72     }
73
74     public Drivetrain getDrivetrain() {
75         return m_drivetrain;
76     }
```

You're now done with your OBIO class modifications. Last time when you were done with this class, you had syntax errors in other files, which pointed out to you exactly what you needed to change next. This time however, since we overloaded the constructor, we don't have those errors. But we still need to change the code that initializes the OBIO because if we don't use our new constructor method, the Drivetrain object won't get initialized. Go to line 32 in RobotContainer.java and update your declaration of the OBIO object accordingly.

```
32     private final OnBoardIO m_onboardIO = new OnBoardIO(ChannelMode.OUTPUT, ChannelMode.OUTPUT, m_drivetrain);
```

Now it's time to create the ToggleLightsBasedOnGyro (TLBOG) command again. For this you can follow all the same steps you did in lesson 6, except there will be some changes to the execute method. To start, follow the steps from lesson 6 exactly. You'll notice that you get an error on the line where you try to get the gyro angle. If you hover over the error, VSCode will explain to you that "m\_drivetrain" is not visible, which is because it was declared as private instead of public, and therefore cannot be accessed from this class:

<pre>// Called every time the s @Override public void execute() {     double gyroAngle = m_io.m_drivetrain.m_gyro.getAngleZ();</pre>	<p>Drivetrain m_drivetrain</p> <p>The field OnBoardIO.m_drivetrain is not visible Java(33554503)</p> <p>Peek Problem (Alt+F8) Quick Fix... (Ctrl+.)</p>
--	---

Thankfully, we prepared for this by creating a getter method for the drivetrain object. Replace the "m\_drivetrain" text with getDrivetrain():

```
double gyroAngle = m_io.getDrivetrain().m_gyro.getAngleZ();
```

Now we don't get an error on m\_drivetrain, but we still have an error, and if you mouse over it you'll see that it's the same error, because m\_gyro is also not visible. Luckily for us, the sample project already created a get method for us. The get method isn't actually for the gyro, but it's for the gyro's Z angle, which is exactly what we want. You can find this method by starting to type "get" after the period after "getDrivetrain()". VSCode automatically checks which methods are available to you and starts listing them out, which can be really handy for quickly finding the method you want to use. This feature is called intellisense. Select the "getGyroAngleZ()" method to complete the line.

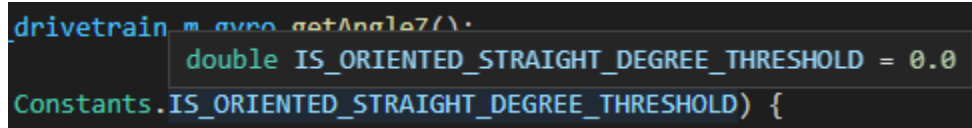
```
public void execute() {  
    double gyroAngle = m_io.getDrivetrain().get  
      
    double gyroAngle = m_io.getDrivetrain().getGyroAngleZ();  
}
```

There's only one more change to make to TLBOG relative to lesson 6. On line 27, you use the number 5 as part of your if statement. This is called a magic number. A magic number is a number that has some real-world logical meaning for your code. In this case it's the number of degrees off of perfectly straight that you're allowing to count as straight. Magic numbers are a necessary evil because you have to use them, but they can be problematic because when reading through code, unless one already understands the significance of the number, the line of code may not make much sense to them. A good measure to reduce the pain of magical numbers is to group them together in a class that is designed purely to hold them. As it turns out, the RomiReference project already has a class designed exactly for this, called Constants.java. Larger projects might want a more specific name than that and potentially more than one class to serve this purpose – for example, AutonomousCalibrations.java, TeleopCalibrations.java, ButtonMappings.java (think back to all the numbers you have to mark down for your axes and buttons), and so forth. But for now we'll just use Constants.java. Grouping magic numbers in a class has a couple of advantages. First, if you need to tune your robot, the tuning levers are conveniently located in a single spot. Right now we only have one value to tune, which is this number of degrees. But we could add others. For example, the scaling coefficients that you use for cut power mode would be well-suited to go in the Constants file as well. After you have a few values like this throughout your program, grouping them all in one place gives you easy access to tune your robot without opening a ton of different files and searching through your code for where you defined them. Additionally, sticking the values in a class lets you replace the number with a textual description of what the number does. This is best illustrated with an example. Here is a replacement for line 27:

```
if (Math.abs(gyroAngle) < Constants.IS_ORIENTED_STRAIGHT_DEGREE_THRESHOLD) {
```

If you read through this code as a human, you might not know what the degree threshold is, but you also might not really care; you can read it and understand that the line of code is checking whether or not the gyro angle is greater than the threshold for whether or not the robot is oriented straight. Unless you're in the process of tuning the robot, you probably care much more about whether or not it counts as straight, than exactly what the cutoff for that condition is. And as it turns out, it's easy to see what the value is if you want to know it anyway, as you'll see in a moment. But first, after you replace line 27 with the code above, let's fix the syntax error. If you click on the red-underlined "Constants", you'll get a lightbulb on the left, and clicking it allows you to import Constants.java. Do so. Now "Constants" turns green but the all-caps text gets a red underline of its own. Click on that and click the lightbulb and it will give you the option to automatically create a constant with that name inside Constants.java (the first option), so click that. Now the underline goes away because it created the constant for you. But, it has no way of knowing what value you want for that constant. Mouse over

“IS\_ORIENTED\_STRAIGHT\_DEGREE\_THRESHOLD” once more and you’ll get a popup showing you the type (double) and value (0.0) of the constant:

A screenshot of a code editor with a dark background. A line of code is visible: `drivetrain.m_gyro.getAngle7().`. A tooltip is displayed over the code, showing the declaration: `double IS_ORIENTED_STRAIGHT_DEGREE_THRESHOLD = 0.0`. Below the tooltip, the text `Constants.IS_ORIENTED_STRAIGHT_DEGREE_THRESHOLD) {` is partially visible.

```
drivetrain.m_gyro.getAngle7().
double IS_ORIENTED_STRAIGHT_DEGREE_THRESHOLD = 0.0
Constants.IS_ORIENTED_STRAIGHT_DEGREE_THRESHOLD) {
```

0.0 is not what we want, but conveniently it’s easy to access and change. If you’re still hovering over the text, simply click, and then press F12, and VSCode will take you to Constants.java and to the line where this constant is declared. Now you can easily change the value to 5, save, and go back to TLBOG. If you mouse over the text one final time, you’ll see that not only is it easier to understand what this line does because it’s written out in text, but you can easily see the value as well because it pops up when you mouse over it.

All you need to do now to finish the project is the same steps you took to finish lesson 6 – set the default command for OBIO and initialize the states of the LEDs, both in RobotContainer.java. After you do these things, deploy your code to check your work! While it won’t do anything differently than lesson 6, your code is much cleaner this time around, and you learned a lot in the process.