Throughout the prior 15 lessons, you've learned an immense amount about the Java programming language, WPILib, and how to program robots. You've learned all the concepts you need to understand the code you see in the RomiReference project and develop your own features. Over the next five lessons, the focus will shift from learning more about Java as a programming language, to developing your skills modifying and writing code. While projects have been developed thus far with clear lists of steps, over the next five projects, the lessons will describe goals and outcomes instead of guiding you specifically on how to accomplish them. There will not be images with exact code solutions, so you will need to use your own skills. There will also not be solution programs provided. Feel free to refer back to prior lessons and projects for both a review of concepts, and examples of how to use certain features of code. We'll step into this slowly. This lesson will still give you some specific pointers, but not all of them, and starting with lesson 17 there will be very few specific pointers. At first this may be difficult for you, but with time and practice you will succeed.

Creating Homemade Drive Methods

So far, you've relied on pre-coded methods to make your robot move, specifically the arcadeDrive method in the DifferentialDrive class, which you access through the arcadeDrive method in the Drivetrain class. You've modified the values that are sent to this method, thereby affecting drive behavior, but you're still using this method to make the robot move. This is fine – the DifferentialDrive class is pretty good for making your robot move. However, as you get more advanced, you'll want a finer degree of control over your robot, and the first step in achieving that is creating your own methods that drive the robot. In this lesson you'll create those. First, you'll create a simple "tank drive" method and associated commands, and then implement your own arcade drive. Tank drive is easier to code because there is less math, whereas for arcade drive you need to convert a rotation value coming from the joystick into power outputs to both wheels. Thankfully you can copy this math from the pre-existing arcadeDrive method in DifferentialDrive, but since tank drive is easier, that's where we'll start.

One quick note – we've briefly discussed deadbanding, or removing joystick input that is close to, but not exactly equal to, zero. The DifferentialDrive class adds deadbanding by default, but as soon as you stop using DifferentialDrive, you won't have deadbanding anymore. You'll implementing deadbanding yourself in the next lesson but keep in mind that your robot might have a tendency to "crawl" whenever it is enabled during this lesson, in particular if your controllers are old and don't do a great job of physically re-zeroing themselves when you release the joysticks. Just watch out and don't let your robot crawl off your desk and fall.

Creating Tank Drive

Tank drive gets its name from tank vehicles, and their drive setups with treads on each side. Each side of the tread can drive independently. Powering them both forward makes the tank move forward, powering them both in reverse drives the tank backward, and powering them in opposite directions makes the tank spin, or if done in a controlled manner, turn. Although you have a Romi and not a tank, any drivetrain that has two independently powered sides of its drivetrain can follow this control scheme, and your Romi of course has that. (Compare this to a front-wheel drive car, where the drive wheels are powered together.)

Tank drive is easy to program because there's basically no math required to do so. Instead of using one joystick axis for robot speed and one for turning, you use one joystick axis to power the left side, and one to power the right side. This makes setting your motor outputs as simple as sending the

input from the joystick directly to the motors. Complete the following steps to create your own tank drive setup:

1. Create a new RomiReference project called “ManualDrive”.
2. In Drivetrain.java, comment out or delete the line where the DifferentialDrive object is declared. This will cause a syntax error later on where that object is used, and you can delete that line as well.
3. Crete a public void method called “tankDrive” that accepts two doubles, one for the left power and one for the right power.
4. In the tankDrive method, simply set the outputs of the two Spark objects using their .set method. Here’s an example of a completed tankDrive method:

```
public void tankDrive(double leftPower, double rightPower) {
  m_leftMotor.set(leftPower);
  m_rightMotor.set(rightPower);
}
```

5. You now have a method that will drive your robot using a tank drive setup, but it’s not being called from anywhere. Right now your drivetrain subsystem runs the ArcadeDrive command by default, which calls the arcadeDrive method in Drivetrain.java, but since you deleted the line of code that told the DifferentialDrive object to drive, that arcadeDrive method no longer has any function and if you were to deploy your robot code, you would not be able to drive your robot in teleop. We’ll fix the arcadeDrive method when we get to the arcade drive portion of this project. For now, create a new command in the commands folder called TankDrive.java.
6. Flesh out your TankDrive command. You can mostly copy the ArcadeDrive command, but change the names of the variables as appropriate. For example, you’re taking in two power variables instead of a power variable and a rotation variable. That’s the only real difference.
7. With the TankDrive command completed, go to RobotContainer.java, find where you set the default command for your drivetrain, and update it to use your new TankDrive command. To do this you’ll need to create a new “getTankDriveCommand()” method in RobotContainer.java. Create this method. You can copy the getArcadeDriveCommand() method and rename it. Note that after creating the method, you’ll still need to update the line that sets the default command, or it will still use the old getArcadeDriveCommand method.
8. At this point you could deploy and drive, but it wouldn’t control very well because you’re still using the default axes for driving. You’ve had to change the default axes (on line 95 of RobotContainer.java) a number of times in various projects throughout the course, and you’ll need to do it again here. The only difference is that you may not want to use the same axes for tank drive as you did for arcade drive. If you’re using a single joystick, there is no comfortable way to do tank drive, so it won’t matter too much which axes you pick. But if you’re using more than one joystick (either two physical sticks, or a controller with more than one joystick such as a gamepad or Xbox controller), you’ll probably want to use the forward/backward axis of two separate joysticks, with the joystick on your left controlling the left speed and the joystick on your right controlling the right speed. You can figure out which axis is which by running your code and using the Joysticks window in the Robot Simulation interface, as you’ve done previously throughout this course. Once you figure out which joysticks to use, update the default joysticks in RobotContainer, deploy your code, and test it. You’ll now have a working tank drive setup!

Creating Arcade Drive

Creating your own arcadeDrive method is a little bit trickier since there's some math involved, but you can copy the math from the arcadeDrive method declared on line 179 of DifferentialDrive.class. Note that there are multiple arcadeDrive methods in DifferentialDrive because it is an overloaded method; the one declared on line 179 is the one you want to look at. The easiest way to open this file is to use F12; you may have an unused import referring to this class roughly around line 10 of Drivetrain.java (it became unused when you deleted the DifferentialDrive object declaration), but if you don't, you can declare another such object, F12 into the class, and then undo the change you made to declare the object. Either way, once you get into DifferentialDrive, the lines you'll want to copy are 199 through 222. We won't discuss exactly how these lines work, but the general gist is that they do a little bit of math to figure out how much power to send to each wheel, based on the speed of the robot and how much turning power to apply. The part you need to recognize is that two important variables and created and initialized by this code: leftMotorOutput and rightMotorOutput. These are the variables you will need to send to the motor outputs. One last thing – since your motors face in opposite directions the way they are physically mounted on the Romi, *one of them needs to run in reverse*, or your robot will spin in circles when you tell it to go forward. In this case, you'll want the motor that runs in reverse to be the right motor. To make it run in reverse, simply multiply its output by -1 before sending it to the motor. Here are a list of steps to complete creating your own arcade drive setup:

1. Open DifferentialDrive.class as described above, and copy lines 199 through lines 222.
2. Paste the copied lines into your arcadeDrive method in Drivetrain.java.
3. You'll have a couple mismatched variable names after pasting the code (for example, xaxisSpeed vs. xSpeed.) It doesn't matter which of these names you choose, but fix the mismatches so you no longer have any syntax errors.
4. At the bottom of your method, *after* all the code you pasted in, update the value of rightMotorOutput by multiplying it by -1.
5. Set the left and right motors to their respective outputs.
6. Test your code and ensure that arcade drive works. When it does, you're done with this project!

Here is an example of a completed Drivetrain.arcadeDrive method:

```java
public void arcadeDrive(double xSpeed, double zRotation) {
  double leftMotorOutput;
  double rightMotorOutput;
  double maxInput = Math.copySign(Math.max(Math.abs(xSpeed), Math.abs(zRotation)), xSpeed);

  if (xSpeed >= 0.0) {
    // First quadrant, else second quadrant
    if (zRotation >= 0.0) {
      leftMotorOutput = maxInput;
      rightMotorOutput = xSpeed - zRotation;
    } else {
      leftMotorOutput = xSpeed + zRotation;
      rightMotorOutput = maxInput;
    }
  } else {
    // Third quadrant, else fourth quadrant
    if (zRotation >= 0.0) {
      leftMotorOutput = xSpeed + zRotation;
      rightMotorOutput = maxInput;
    } else {
      leftMotorOutput = maxInput;
      rightMotorOutput = xSpeed - zRotation;
    }
  }

  rightMotorOutput *= -1;

  m_leftMotor.set(leftMotorOutput);
  m_rightMotor.set(rightMotorOutput);
}
```