Final Project Directions

This robot project already has a mostly-working drivetrain simulation class. You should start with fixing the errors in that class, but the other tasks do not need to be completed in an exact order. You may not be able to finish all of the tasks, so if you get stuck, try working on another. If you do finish the earlier tasks, try taking on the challenge tasks. If a challenge task requires changing something about the code for an earlier task, comment out your previous code instead of overwriting it.

Drivetrain Tasks

- Find the three errors in `Drivetrain.java` and fix them. Try running the code to see if you caught all of them.
- Line 106 (prior to making any changes to the file), inside the `resetOdometry` method, is commented out. Uncomment the code and add the correct parameters to the method call, using documentation as needed. If your fix works, pressing the 1 key should reset the robot's simulated position and orientation to where it starts at.
- The max speed of the drivetrain is currently fixed at 4 m/s. This can sometimes be a bit difficult to control though, so create a public `setMaxSpeed(double maxSpeed)` method that will change the max speed of the robot. Use this method to set the robot to a slower speed when you press the b button, and then set it back to the full speed when you release it. Use the simulator to check that the robot drives more slowly when holding the 1 key and that it goes back to full speed when you release it.
    - Challenge: Holding down the key can be a bit annoying in the simulator. Make it so the speed is toggled when you press the key instead of having to hold it down.
- It's often useful to know how fast our robot is traveling. We can assume that the xSpeed value passed into the `drive()` method is close enough to the robot's actual speed, so track this value in a new subsystem field. Update it and log its value to SmartDashboard whenever it is set in the `drive()` method, and create a public `getSpeed()` method that returns its current value.

Intake Tasks

- There's currently an intake class, but it doesn't do anything! Add methods to set the intake motor speed, and bind them to the x and y buttons (keys 3 and 4 in sim) so the intake runs forwards with a positive speed of 0.1 when the x button is held and backwardw with a negative speed of -0.1 when y is held. The intake should only run while the buttons are held. You can plot the `Intake Motor Speed` field in the simulation to make sure this method works.
- Some intakes can also be pulled into the robot when they aren't in use to protect them from damage. The easiest way of doing this is to use pneumatic pistons which can be commanded to a fully extended position or a fully retracted positions. This is what our intake will use.
    - All of the pneumatics on the robot are controlled by a single PneumaticsControlModule. Construct a `PneumaticsControlModule` in `RobotContainer.java`.
    - To control our intake, we'll need two solenoids that control the position of the pistons. When constructing the pistons in `Intake.java`, use a `PneumaticsModuleType.CTREPCM` for the moduleType parameter. Instead of hardcoding the module channel numbers, create a constructor for `Intake.java` that takes two integer parameters and uses these values when constructing the solenoids. If you don't know how to do this, just hardcode the values of 5 and 6. Also construct an `Intake` subsystem in `RobotContainer.java`.
    - We'll assume that the setting the solenoids to true will extend the intake, and setting them to false will retract it. Create public methods to extend, retract, and toggle the piston states. Log the state of the pistons using SmartDashboard methods.
    - Whenever the intake is set to run in forward or reverse, extend the intake. Whenever the intake is stopped, retract it.
        - Challenge: Instead of modifying the code in your `Intake` class to acheive this, try using Command Compositions to do this in `RobotContainer.java`.
- Challenge: For some gamepieces, the optimal intaking speed actually depends on how fast the robot is moving. Use the `getSpeed()` method you wrote in your drivetrain class to set the intaking speed to a quarter of the robot's speed. Make sure this value is always positive.

Extra Challenge

- If you finish all of the other activites, try to make an autonomous routine for your robot. This will entail creating a command that drives the robot in a predetermined path, perfoming functions like intaking along the way. The command compositions that you have already worked with in other challenge activites will be the easiest way to accomplish this. To drive for a set time or distance, you'll want to use a `WaitCommand` or a `ConditionalCommand`. Return the constructed autonomous command sequence from the `getAutonomousCommand()` method at the bottom of `RobotContainer.java`. To run the routine, select 'autonomous' from the robot state menu in the simulator instead of 'teleoperated'.