In this lesson we'll focus on a couple more critical OOP concepts, and finally cover the keyword "extends", which you've seen when working with commands but haven't had explained to you yet. We'll do a quick review of a few classes in your RomiReference project so you understand how the concepts relate, and then do a desktop project without the Romi to get some practice.

Composition, Inheritance, & Polymorphism

There are three main concepts to cover right now: composition, inheritance (extension), and polymorphism.

Composition is pretty simple and you've already seen a number of examples of it. It's the idea that classes can be *composed* of other classes. You use this every time you declare an object of as a class field in an object of another type. For example, here's a snippet from the Drivetrain class in lesson 11:

```
16  public class Drivetrain extends SubsystemBase {
17    private static final double kCountsPerRevolution = 1440.0;
18    private static final double kWheelDiameterInch = 2.75591; // 70 mm
19
20    private double targetHeading;
21    Timer turnTimer = new Timer();
22
23    // The Romi has the left and right motors set to
24    // PWM channels 0 and 1 respectively
25    private final Spark m_leftMotor = new Spark(0);
26    private final Spark m_rightMotor = new Spark(1);
```

Among other objects, the Drivetrain class is composed of a Timer object and two Spark objects, representing the timer we use for driving straight, and the left and right motors. Composition is simple but powerful as it allows for complexity to be broken into simpler pieces, and for code reusability as classes tend to be self-encapsulated units that can be used in more than one place.

Inheritance is the idea that you can define a class (called a "subclass" or "child" class) which has all the characteristics of its "parent" class, and then additional characteristics as well. If you have a lot of classes which share common functionality, this is useful, because that way you don't need to re-write the same functionality. You've already used inheritance too although you probably haven't realized it. Every time you see the keyword *extends*, you're using inheritance. A class that extends another class, inherits attributes from that class. In the image above, Drivetrain extends SubsystemBase. SubsystemBase is a class the CommandScheduler recognizes as a subsystem. So when Drivetrain extends SubsystemBase, Drivetrain is also recognized as a subsystem, which allows you to run commands that require it, such as the autonomous commands you wrote in lesson 11 and also the ArcadeDrive command. Drivetrain also literally does *extend* the SubsystemBase class, because you add a bunch of functionality. When you need a Drivetrain class, you need a Drivetrain class – not any random Subsystem. If you got some SubsystemBase object that didn't have, for example, left and right motors, it wouldn't do you any good.

Polymorphism is a fancy word for an important concept. It basically means that an object of a certain class can exist as a few different kinds of objects. Let's again use our subsystems and commands

as examples. The CommandScheduler runs your commands by calling their execute method constantly. It knows that it's calling the execute method of a command. However it does not know if it's calling the execute method of an ArcadeDrive command, a DriveDistance command, a TurnDegrees command, or something else. From its point of view, it's simply calling the execute method of a Command object. In reality that Command object might be of any of the types of command just listed, or something different. This is polymorphism: all the different types of commands are each, well, commands, and so a generic Command object could end up being any of those types, and that's fine since they all have the same initialize, execute, end, and isFinished methods. This will make more sense as you see more examples of it.

That's a brief introduction to these concepts. For a deeper study including lots of examples of code, read sections 1-4 (NOT 1.1-1.4, the full sections 1-4) on this page. You may want to open the code in a desktop app in VSCode and follow along, since the text portion is fairly short and they rely on code examples:
https://www3.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

Examples in the RomiReference Project

Now that you've learned more about these concepts, let's see how they apply to our robot code. Open a default RomiReference project, open DriveDistance.java, and look at line 10. You'll see the text "extends CommandBase". You learned before that the extends keyword is how Java knows your command is actually a command, and now you understand why. CommandBase is a command, so any class that extends CommandBase is also a command. A command is defined as having the four methods we've talked about – initialize, execute, isFinished, and end – so DriveDistance has these methods, and they're tagged with @Override as explained in the reading. We'll talk about *how* a command is defined as having these four methods in an upcoming lesson.

There are numerous similar examples in the RomiReference object but let's look at just one more. Open Robot.java. In a previous lesson you learned about how there are methods in this class that run constantly depending on what mode the robot is in (disabled, teleoperated, etc.) There's also the robotPeriodic() method that runs all the time regardless of which mode you're in. Why do these methods run all the time? Look at line 17 where the class is defined:

```
17   public class Robot extends TimedRobot {
```

The Robot class extends TimedRobot. If you click on TimedRobot and press F12, you'll be taken to the class definition for TimedRobot. You'll notice that TimedRobot extends IterativeRobotBase:

```
20   public class TimedRobot extends IterativeRobotBase {
```

This is a good example of how you can make a *hierarchy* by extending classes. Your Robot class extends TimedRobot, which extends IterativeRobotBase. Therefore your Robot class *is* a TimedRobot, and is also an IterativeRobotBase. This allows you to go from general to specific. In the case of the Romi, the general classes are all written for you and wouldn't modify them unless you're doing something quite advanced. When you write your own programs from scratch, you'll create your own hierarchies.

For now, if you once again F12 on IterativeRobotBase, you'll again be taken to a class definition. In this file, you can ctrl-f and search for "periodic", and you'll find all of the period methods we were just talking about. Don't worry about most of the code you see in this file, but let's take one quick peek at

the teleopPeriodic() method:

```
179     /** Periodic code for teleop mode should go here. */
180     public void teleopPeriodic() {
181       if (m_tpFirstRun) {
182         System.out.println("Default teleopPeriodic() method... Override me!");
183         m_tpFirstRun = false;
184       }
185     }
```

On line 182 above, you'll notice there's an SOP saying "Override me!". When the people writing the WPILib were making these general classes that get extended, they included this line so that if you run code without realizing you need to override these periodic methods, when you run your code the default, or super, method will be called and it will tell you to override it. In the RomiReference project these methods are already overridden – in this case on line 85 of Robot.java – but this is just an example of how the hierarchy works. Looking through the code like this gives you a deeper understanding of exactly how your Java program works.

RobotComposition Project

When modifying an already-existing RomiReference project, most of the hierarchy work is already done for you. It's important to understand it, and when doing advanced work with your robot you will even add to it. However, for solidifying your understanding of these concepts, it's best to make your own hierarchies from scratch, so we'll do a desktop project for this lesson. In this project you'll create the hierarchy for defining robots that can participate in various FIRST Robotics Competition (FRC) games. If you aren't familiar with FRC, here are some links to YouTube videos that show how the games work. FRC robots are programmed using the same exact WPILibrary that you're using to learn programming, so if you have the opportunity to join a team at your school or in your community, not only will you have bigger and stronger robots to play with than the Romi, but you'll already be ready to dive right into programming them. Here are the videos for the 2018, 2019, and 2020 editions of the competition. Pay attention to the possible different mechanisms on the robots. They all have drivetrains, along with other mechanisms that depend on the robot and the game. Some robots might have elevators, some might have ball shooters, and some might have climbers. You don't need to worry about the point values.

- 2018, Power Up: https://www.youtube.com/watch?v=HZbdwYiCY74
- 2019, Destination: Deep Space: https://www.youtube.com/watch?v=Mew6G_og-PI
- 2020, Infinite Recharge: https://www.youtube.com/watch?v=gmiYWTmFRVE

Now that you've seen the games, let's define a hierarchy for *composing* some robots in code. For the sake of this project, we will assume that robots from the following games have the following types of manipulators. All robots will have drivetrains:

- Power Up: a climber and an elevator.
- DeepSpace: a climber and an elevator.
- InfiniteRecharge: a climber and ball shooter.

Now, for the project:

1. Create a new desktop project called RobotComposition and then create the following classes:
    a. RobotComposition, which will contain your main method
    b. Robot
    c. Drivetrain
    d. GamePieceManipulator
    e. CubeElevator
    f. BallShooter
    g. HatchPanelPlacer
    h. PowerUpRobot
    i. DeepSpaceRobot
    j. InfiniteRechargeRobot
2. In your robot class, define two *protected* class fields, called "drivetrain" and "manipulator". Make their variable types "Drivetrain" and "GamePieceManipulator", respectively. We haven't talked about *protected*, but it is in between public and private. With protected variables, other classes cannot access them, but subclasses can, making them handy for use when creating hierarchies.
3. In your PowerUp, DeepSpace, and InfiniteRecharge robot classes, use the **extends** keyword to make these classes inherit from the class "Robot". We will refer to these three classes – but not Robot itself - as "game robot classes".
4. Make your CubeElevator, HatchPanelPlacer, and BallShooter classes extend the class "GamePieceManipulator". We will refer to these three classes – but not GamePieceManipulator itself - as "manipulator classes".
5. In each of your game robot classes, create a constructor method that has no parameters. Inside your constructor, create an object of the class of the manipulator that best corresponds to that game. Then set the game robot's manipulator class field (*which it inherited from Robot, and therefore you don't see in the code file you're looking at*), to the newly created object. Note how the "manipulator" variable is of type GamePieceManipulator, but across our three game robot classes, we were able to set it equal to objects of three different classes. That is because all of these classes, despite being different from each other, are still GamePieceManipulators.

    Example: DeepSpaceRobot constructor. Note how we can set this.manipulator, despite it not being defined inside of the file.

```
src > DeepSpaceRobot.java > ...
1   public class DeepSpaceRobot extends Robot {
2       public DeepSpaceRobot() {
3           HatchPanelPlacer placer = new HatchPanelPlacer();
4           this.manipulator = placer;
5       }
6   }
```

6. Go into GamePieceManipulator and create a method called "manipulate" that SOP's some message.
7. Go into Robot and create a method called "manipulateGamePiece calls the manipulator.manipulate() method.

8. In your main method, create a robot object and a GamePieceManipulator object. Set the Robot object's manipulator variable equal to your newly created GamePieceManipulator object. Then call your robot's manipulateGamePiece method. Now is a good chance to run your code and check that everything is working – if it is, you'll see the SOP line you added to GamePieceManipulator.manipulateGamePiece get called and you'll see the output in your terminal.

```
1   public class RobotComposition {
        Run | Debug
2       public static void main(String[] args) throws Exception {
3           Robot robot = new Robot();
4           GamePieceManipulator manipulator = new GamePieceManipulator();
5
6           robot.manipulator = manipulator;
7           robot.manipulateGamePiece();
8       }
9   }
```

9. Go into your three manipulator classes and create manipulate() methods there. These methods should output something identifying what kind of manipulator they are, e.g. "now shooting power cells."
10. In your main method, create three new objects, one for each type of robot.
11. Call the manipulateGamePiece method for each of your robot objects. Example:

```
robot.manipulateGamePiece();
powerUpRobot.manipulateGamePiece();
deepSpaceRobot.manipulateGamePiece();
infiniteRechargeRobot.manipulateGamePiece();
```

You will have four lines of output, each of which is a message describing a type of manipulator. Note a couple things:

1. We defined a "manipulate" method for Robot.java so all robots can "manipulate', even if the output is generic.
2. We overrode "manipulate" in our manipulator subclasses, so when we called .manipulate() in Robot.java, it was able to perform more specific functionality. (Polymorphism.)
3. In Robot.java, we created a manipulateGamePiece() method, which we never overrode. Since we extended Robot.java with our robot classes, they already had access to the manipulateGamePiece() method, despite us never defining it in those classes.