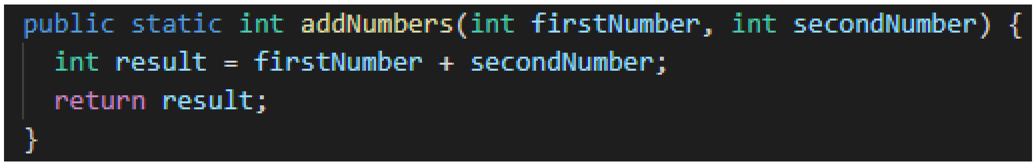
In this lesson we round out the last couple small concepts of OOP that we've skipped thus far and do another code review of some Romi code because after this lesson you'll understand every concept that you need to, to grasp everything that goes on in the Romi code, at least from a Java perspective. (You will still run into WPILib methods that you might need to check the documentation on – but at least you'll understand all the Java code that they're made from.) After the code review you'll do some small desktop projects. After this lesson, the remaining lessons in the course will all be advanced projects with the Romi that will give you a chance to flex your knowledge and understanding. The two concepts left to cover are *static* and *super*.

Static

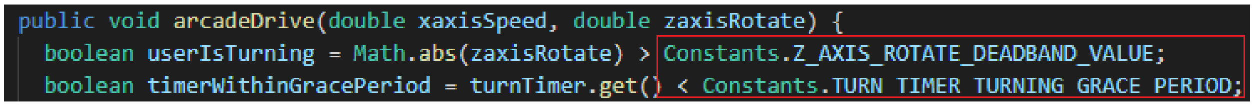
One of the main concepts in object-oriented programming is that objects have state. That means there exists some object of some class, it has some variables, and cumulatively they define that object. If you have multiple objects of a given class, they probably have different values in their variables, thus differentiating them. If you were to access the variables of the objects directly, you would get a different value depending on which object you accessed. If you were to call a method of the object, you would again get a different value depending on which object you called the method for, because the variables would have different values. E.g., if in a fantasy game if you called getLevel() on an orc, you might get a different result than if you called getLevel() on a dragon.

However, there are sometimes instances where a class field or method will return the same result regardless of which object it came from. For example, consider the following method:

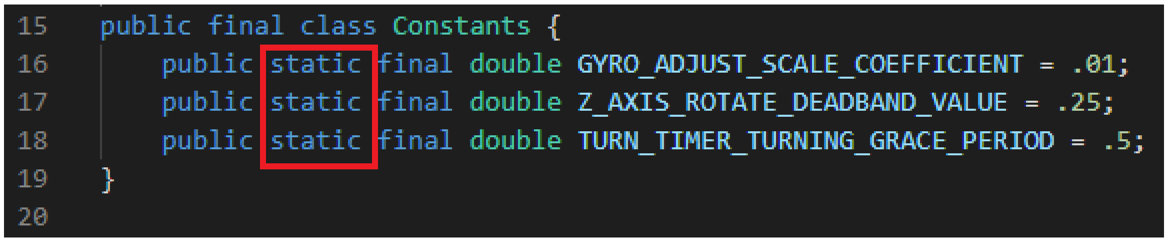

```java
public static int addNumbers(int firstNumber, int secondNumber) {
  int result = firstNumber + secondNumber;
  return result;
}
```

This method could exist in any class, but no matter where it exists, or the state of any objects belonging to the class where it exists, it will always return the same result for any given numbers as input. Nothing else about the class it exists in, or any objects of that class, matter – it will simply add the two numbers and return the result. *The state of the object that the method is being called from does not matter.* When a method or variable fits this description, you can use the *static* keyword in its signature. It's not mandatory to do so – you could remove the static keyword from the method above and the method would still function the same way. However, the static keyword grants a special ability – *you can call the method without instantiating an object of the class.* This can be illustrated with a couple examples because you've already used this functionality numerous times during this course. Think about every time you've used System.out.println(). System is a class, but you've never declared a System object. Turns out that the *out* portion of "System.out" refers to a static object, so you can use the functionality of System.out without declaring an object. This is handy, because it would be annoying to declare System objects every time you wanted to do something as simple as output a line of text.

Another example is the Constants file you've used in your Romi code. Here are a couple lines from the ArcadeDrive method in your DriveStraight project (Lesson 11 version):


```java
public void arcadeDrive(double xaxisSpeed, double zaxisRotate) {
  boolean userIsTurning = Math.abs(zaxisRotate) > Constants.Z_AXIS_ROTATE_DEADBAND_VALUE;
  boolean timerWithinGracePeriod = turnTimer.get() < Constants.TURN TIMER TURNING GRACE PERIOD;
```

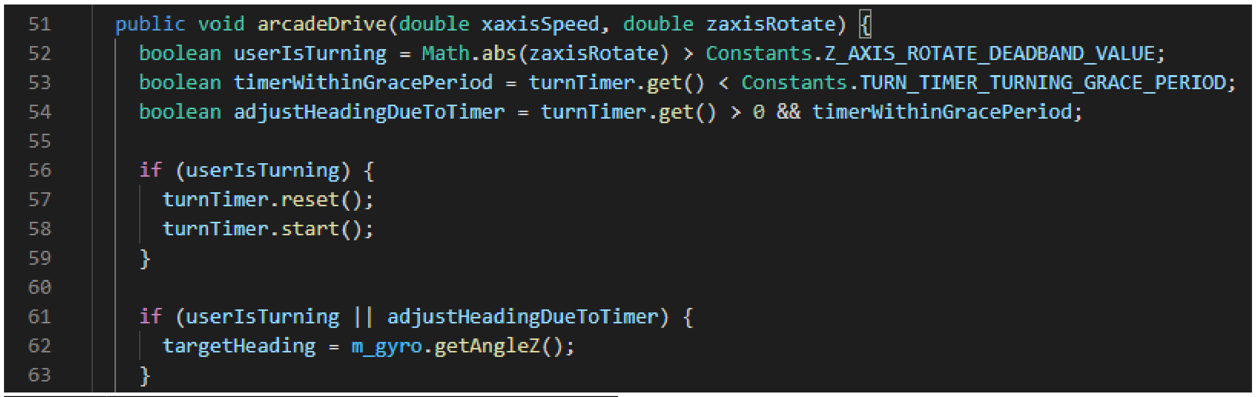
These lines reference values from the Constants file, but you never create a Constants object. Open Constants.java and you can see that these values are declared statically:



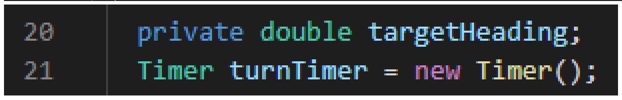
```
public final class Constants {
    public static final double GYRO_ADJUST_SCALE_COEFFICIENT = .01;
    public static final double Z_AXIS_ROTATE_DEADBAND_VALUE = .25;
    public static final double TURN_TIMER_TURNING_GRACE_PERIOD = .5;
}

```

The static declaration lets these constants be accessed anywhere without declaring a Constants object. This is possible because, as static constants, they do not rely on the state of any object – they will always return the specified value.

Static works for both methods and values, as shown in the examples above. The only factor is that anything declared as being static can't rely on the state of the class in which it's declared. Let's look at a counterexample of this to demonstrate the point. Here are some lines the ArcadeDrive method from the DriveStraight project again:



```
51  public void arcadeDrive(double xaxisSpeed, double zaxisRotate) {
52    boolean userIsTurning = Math.abs(zaxisRotate) > Constants.Z_AXIS_ROTATE_DEADBAND_VALUE;
53    boolean timerWithinGracePeriod = turnTimer.get() < Constants.TURN_TIMER_TURNING_GRACE_PERIOD;
54    boolean adjustHeadingDueToTimer = turnTimer.get() > 0 && timerWithinGracePeriod;
55
56    if (userIsTurning) {
57      turnTimer.reset();
58      turnTimer.start();
59    }
60
61    if (userIsTurning || adjustHeadingDueToTimer) {
62      targetHeading = m_gyro.getAngleZ();
63    }
```

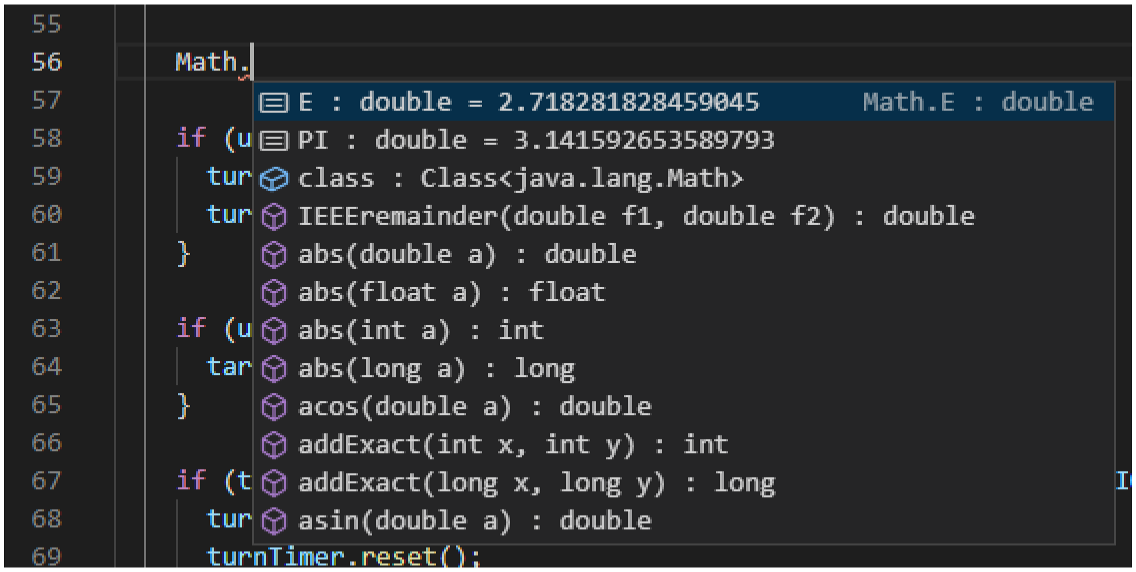


```
20    private double targetHeading;
21    Timer turnTimer = new Timer();
```

Note how on lines 53, 54, 57, and 58, the method relies on turnTimer, which is a class variable declared on line 20 in Drivetrain.java. A static method can neither read nor write to such a variable, because doing either of those things requires a class state – either by reading an existing state or creating a state by writing one. That could not be done without an existing Drivetrain object. Another example is on line 62, arcadeDrive updates the value of targetHeading, which is another class variable. Neither of these operations could work without maintaining the state of these variables, so arcadeDrive could never be a static method.

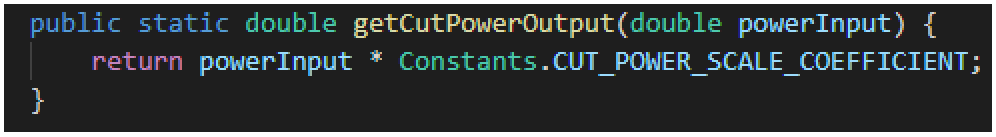
You've probably observed by now how class names are capitalized, like Constants, Drivetrain, and System, but variable names start with lowercase letters. This makes it easy to tell, for any given text snippet, if it's referring to a variable or a class itself. If you look at lines 52 and 53 in the image above, you'll notice that "Constants" is capitalized – this is because it's referring to the class itself. Static fields and methods are accessed in this manner, using the proper class name instead of a specific object.

Another fun example of static methods and variables in Java is the "Math" class. Anywhere in your code, you can type "Math." and see a big list of static methods and values you can use, for example Math.PI or Math.abs, which you've used before.



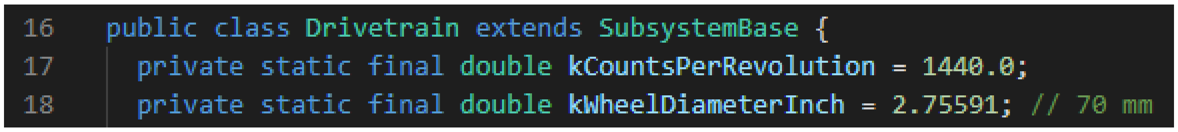
```

    Math.
        E : double = 2.718281828459045        Math.E : double
    if (u PI : double = 3.141592653589793
      tur class : Class<java.lang.Math>
      tur IEEEremainder(double f1, double f2) : double
    }     abs(double a) : double
          abs(float a) : float
    if (u abs(int a) : int
      tar abs(long a) : long
    }     acos(double a) : double
          addExact(int x, int y) : int
    if (t addExact(long x, long y) : long                        I
      tur asin(double a) : double
      turnTimer.reset();
```

As a general rule, static values are great for declaring program-wide constants, which is literally why the Constants.java file exists in the RomiReference project. Static methods are great for helper methods that perform operations on inputs. For example, you could have made a static method when you did the CutPower project that would have looked something like this:



```java
public static double getCutPowerOutput(double powerInput) {
    return powerInput * Constants.CUT_POWER_SCALE_COEFFICIENT;
}
```

That may or may not have been overkill for such a simple operation, but it demonstrates the point – this method could be called from anywhere in the program because it simply performs an operation and it doesn't rely on the state of anything in your program. Let's look at one final example from robot code, in Drivetrain.java:



```java
16  public class Drivetrain extends SubsystemBase {
17    private static final double kCountsPerRevolution = 1440.0;
18    private static final double kWheelDiameterInch = 2.75591; // 70 mm
```

These values are declared statically because they describe physical characteristics about the robot. For example, line 18 is telling the code what the diameter of the Romi's wheels is, which is used in calculations for how far the Romi has driven, but never changes and while it is related to the drivetrain, does not rely on anything in the Drivetrain class because it's simply a physical truth about the robot.

Super

Using inheritance creates a scenario where for a given class, there can be more than one version of a given method. One version of the method can belong to the subclass, and one can belong to its parent class, or "superclass". In fact there could be many versions, as there could be a long chain of inherited classes, and every class in the chain could have their own version. The same situation can happen with constructors – you know that a constructor gets called whenever an object is created, but, looking at your project from last lesson, if an object of type “Player” is created, and Player objects are also “Character” objects, which constructor gets called?

Java has default behavior to handle this situation but also gives you the ability to control what happens more directly. When a method is called for an object, it will use the method defined for the subclass. In a long chain, it will use the lowest-level class it can (that is, if the lowest-level class does not have that method defined, it will use the direct parent’s version. If the direct parent does not have that method defined, it will use the grandparent’s, etc.) This gives you the power to override parent class methods but use them as defaults if you don’t want to override them.

However, parent versions of methods are accessible from within a subclass by using the *super* keyword. The super keyword is *not* a variable, but it almost acts like one – if you’re in a class method (go into your Player class, inside a method, and try this) and you type “super.”, VSCode will pull open a menu of methods that you can call, the same as if you typed the name of an object variable that has methods. From super, you can find any method defined in your parent class and call it directly.

The same syntax can be used for constructors. Every time you create an object using the “new” keyword, you call a constructor. Also remember that every class has a default constructor method, even if you don’t define it. Remember, a “default” constructor is one that takes no parameters. When an object of a child class is created, you call the child class’s constructor using the new keyword. Java will then automatically call the superclass’s default, or parameterless, constructor. This is referred to as calling the constructor “implicitly”. Often such a constructor does nothing so you would never notice, but you can try this out. Go into your Character class from lesson 14 and add the following method:

```
public Character() {
        System.out.println(“Java called this constructor automatically!”);
}
```

If you run your program, you will see this printout for each of the character objects you created, despite you never manually calling this constructor. Since you called your subclass constructor, and Java implicitly called the superclass constructor, that means that for subclass objects, *more than one constructor is called.*

By default Java will call the parameterless superclass constructor. But if you have custom constructors defined in your superclass and want to call a specific one, you can again use the super keyword. In this case, super is the name of a method and you give it parameters directly – for example, “super(name, hitPoints);”. If you find that you have similar constructors among your different subclasses, creating a superclass constructor and then referencing it from the subclasses can be a good strategy. The call to the superclass constructor goes in the subclass constructor, which can then do other things. For example, if all Character subclasses need to set the character’s name, you could create a Character superclass constructor to do that. But then maybe Enemy class objects need some additional work – say, initializing a “difficultyLevel” variable. In your Enemy class constructor, you would call super(name) to set the name, and then set the difficultyLevel like normal.

--

1. Create a new desktop project and a class called “BasicOperations”. Then define and implement the following four **static** methods. Each class is to take two integer parameters and return and integer, except for “divide” which will take two doubles and return a double.
    a. add()
    b. subtract()
    c. multiply()
    d. divide()
2. In your main method, call each of your newly created methods in a static fashion. That is, do not create an object of the class type BasicOperations – simply call the static methods.
3. Call the following methods from the java “Math” class (e.g. double varName = Math.cos(10);) and output their values:
    a. Math.cos
    b. Math.sin
    c. Math.PI (this is a constant, not a method)
    d. Math.random (read the description VSCode gives to see what this method does)
4. In the lesson 14 exercises, you created constructors for Player, NPC, and Enemy that all called the Character.setName method. Instead of using these constructors, create a single constructor in the Character class that accepts a String parameter and sets its name variable. Then, in your subclasses, call this new superclass constructor.