Intro to FRC Control

Team 359 - Meilan Muto

Foreword

This document will attempt to serve as a complete introduction to programming for control systems on team 359. As technical specifications such as language syntax and mathematical formulae would be better explained through online tutorials, the focus of this document will be placed primarily on the conceptual. Of course, some important technical information will be included for readability's sake. You may interpret this document as a collection of ideas and strategies that are un-abstracted to real-life applications in the realm of FRC robotics. This document will cover problem-solving in control, basic control theory, introduction to Java, and source control.

Note, however, that as I write this document, I am still a student in the program and lack the knowledge and experience to provide in-depth explanations or professional insights into programming, or even FRC robotics. Rather, this document will consist of my personal experience with robotics and the things that I, a three-year student, have acquired through this program. Consequently, there may be errors and unspecificity in any technical details that I may inevitably provide. Please be mindful of this as you read.

The recommended prerequisite for this document will be fluency with basic programming concepts such as fundamental loops, conditionals, and code flow. However, this document will attempt to remain accessible for readers of all levels, from complete beginners to relatively experienced programmers.

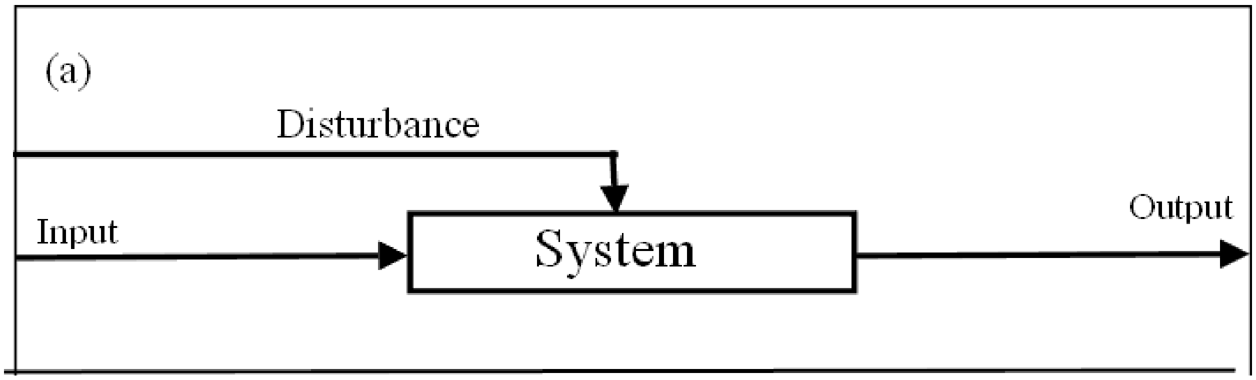
What is Control Systems?

First and foremost, what is ‘control systems’?

Within team 359, when people mention ‘Control systems’, they are usually referring to the subdivision of the Robot Construction team tasked with taking robot components and making them function in desired ways. This includes the electrical wiring of the robot, the pneumatic ‘plumbing’ of the robot, and the programming of the robot. As previously mentioned, this document will focus specifically on the programming aspect of the process.

However, in the context of robotics and engineering as a whole, the term' control system' typically means the system that controls a mechanism or sets of mechanisms in their interaction with the physical world. In generalizing the definition to encompass a broader concept, this particular definition seems to have lost its specificity, so this introduction will clarify the term control system and introduce some examples and sub-concepts.

Unlike code in a virtual setting, physical machines have physical limitations and imperfections that can cause the machine to behave in unforeseen ways. In the case of a robot arm, for instance, these physical limitations may include but are not limited to, friction, gravity, inertia, and even wind resistance. Basically, we can’t simply command a motor to move a specific amount and expect the mechanism to always end in that specific place. This is why control systems are used.



A system encounters disturbances that adversely affect its ability to perform a task

In essence, control systems bridge the gap between theoretical instructions and physical implementations. By applying things that are understood about a machine’s environment to the theoretical instructions, we hope to achieve greater accuracy in the physical implementation.

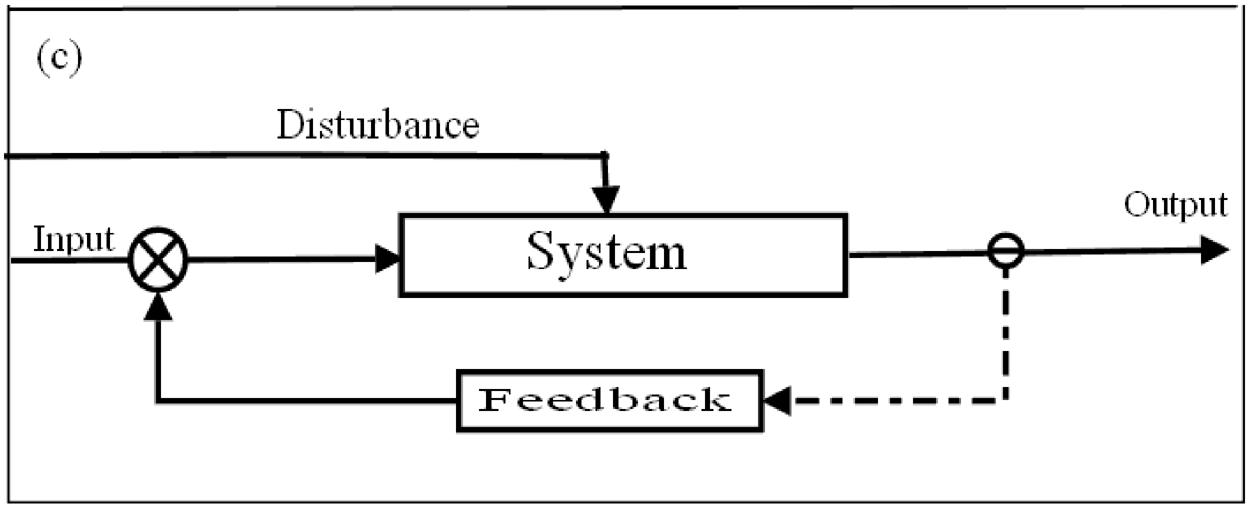
Some keywords to keep in mind:
- State - the state of being
- System - the object being controlled
- Load - the stresses that are imposed on a system
- Environment - the space surrounding a system of interest
- Error - the difference between the desired and current state of a system

Feedback

By and large, feedback control is the most frequently implemented control scheme. Feedback controls are based on the constant repetition of two steps: Sensing and commanding. Feedback controllers are used to compensate for unexpected interferences that may impede a system from performing its task.

Sensing is primarily concerned with sensing load on a system and compensating for its interferences. If you have a background in robotics of any kind, you may be familiar with sensors such as gyroscopes, accelerometers, distance sensors, and encoders. These are all devices that are used to conveniently understand the state of things in and around a robot. Useful information is periodically extracted from sensors, processed by filters or operators, and archived in program memory where it can be used in the command process.

Commanding is primarily concerned with altering the state of the system. Often, the command-period integrates or converts outside instructions with processed sensor readings to produce a new command for the system.



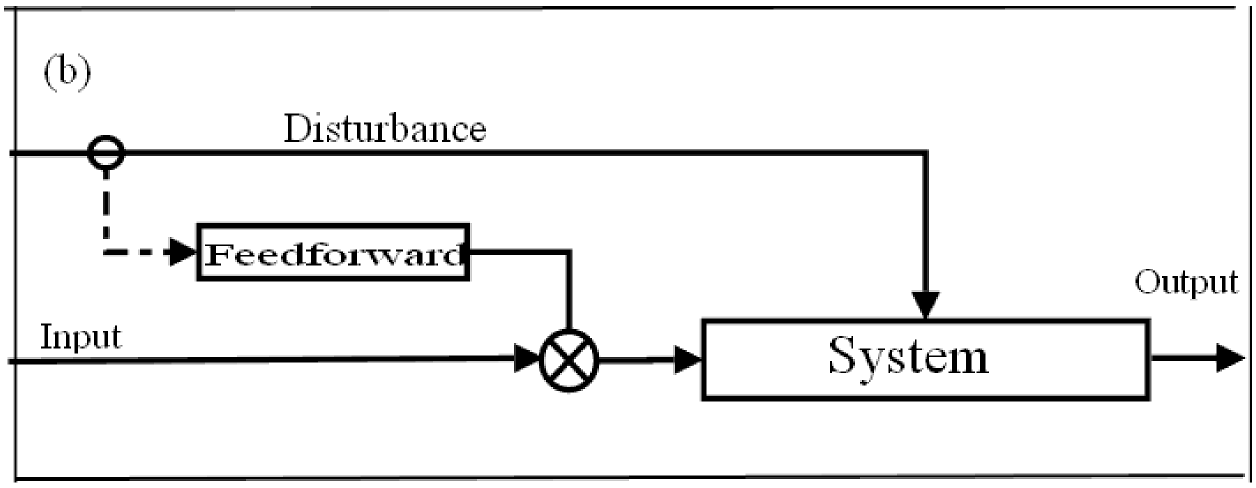
A system encounters disturbances that are measured and 'fed' back for compensation

In Vex, you may have used premade commands that moved a motor to a certain position. This command, and many others like it, were created using feedback control. More specifically, a subset of feedback control called a PID controller, covered in greater detail here. Every loop iteration, an error value is generated using the encoder position and desired position, and depending on the size of this error, the power of the motor is increased or decreased. In order to prevent overshoot, one might program the arm such that while the arm is far away from its destination, the power is higher, and as the arm approaches its destination, the power decreases proportionally. By sensing the position of the arm with the encoder and using the error to adjust the speed, we have created a simple feedback controller that drives an arm to its desired position.

Feedforward

The fundamental flaw with feedback controllers is their latency. By definition, the feedback controllers are reactionary or adaptive. When interferences are introduced to a system, a feedback controller take time to compensate, and in applications where stability and accuracy are crucial, the latency of feedback controllers may have noticeable impacts. Feedforward controllers are, as the name implies, controllers that feed information forward or anticipate interferences.

In feedforward control, known stresses on the system are determined prior to run-time and compensated before the system accuracy is affected. Like feedback controllers, feedforward controllers can utilize sensor information; But, unlike feedback controllers, feedforward controllers also utilize predetermined models for system behavior. Regardless of their source, feedforward controllers typically use the information to anticipate stress on the system and counteract them, negating their effects in real-time. The ultimate goal of feedforward controllers is to minimize the workload for feedback controllers by accounting for the know stresses.



A system is able to anticipate a disturbance and counteract it in real-time

A simple example of a feedforward controller is one utilized to counteract static thresholds. You may be aware that there is a region at the lower end of a drive motor's power output in which the robot does not move. Though the primary reason lies in the mass of the robot, there are several factors that contribute to this effect including slippage between the wheels and their axles. If, however, we knew through experimentation that the robot began moving at a percent output of 12% of full power, we could simply add this value to all incoming commands and map the remaining 82% power to values on the joystick. The joystick would become far more responsive and the initial 12% of the joystick that was wasted on meaningless control would finally see some use.

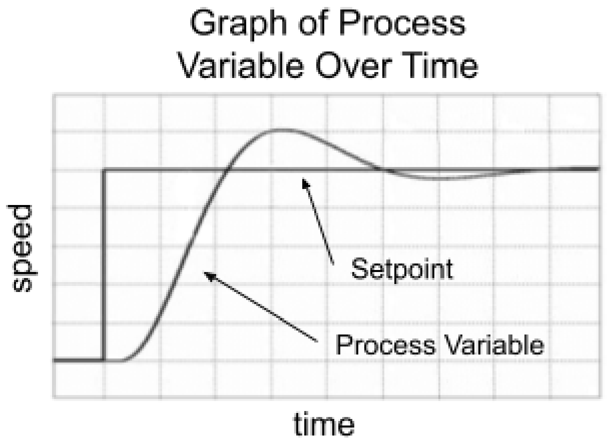
Similarly, feedforward controllers are often used for gravitational resistance on elevators and arms. Through simple experimentation or calculation, it is easy to predetermine the force required for the motor to counteract gravity and consistently apply this force to the arm. This way, an idle arm will be able to stay in the same position without consuming needless processor time.

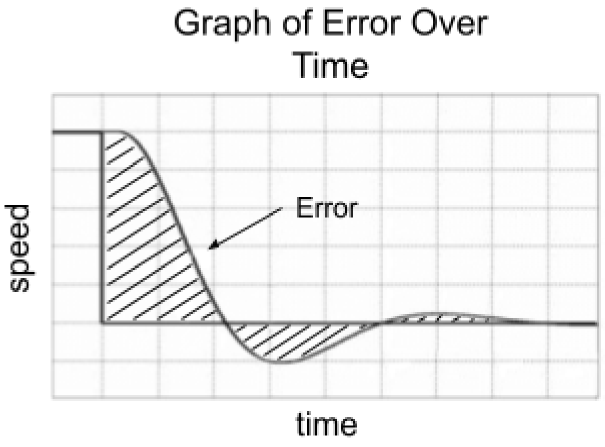
PID controllers

PID controllers are an extremely, if not the most, important concept to understand in all of FRC control systems. PID controllers — or Proportional, Integral, and Derivative controllers — are the fundamental unit of control for most FRC mechanisms, so further self-exploration is highly recommended.

Some terms to know:

- Process variable - the current value of the system
- Setpoint - the desired value of the system
- Gains - constant scaling terms
- Error - the difference between the setpoint and process variable
- Integral - the area under a curve
- Derivative - the rate of change of a curve at time t





Flipping the graph of a process variable and setpoint with respect to time gives a graph of error with respect to time

The equation for a PID controller looks like this:

$$K_p e(t) + K_i \int_0^t e(t)dt + K_d \frac{de}{dt}$$

This may seem quite daunting at first, but rest assured, this equation is much simpler in practice. I'll also add that for our purposes, it is typically unnecessary to use all PID terms, and often, we will use a simple P-controller or PI-controller, omitting the derivative term altogether.

The proportional term of a PID controller represents the error at t, scaled by a proportional constant Kp. This means that when the error is large, so will our controller's output.

The Integral term of a PID controller represents an accumulation of error through a system's entire run-time scaled by an integral constant Ki. This means that the integral output will grow larger if the system fails to reach its desired output in a timely manner.

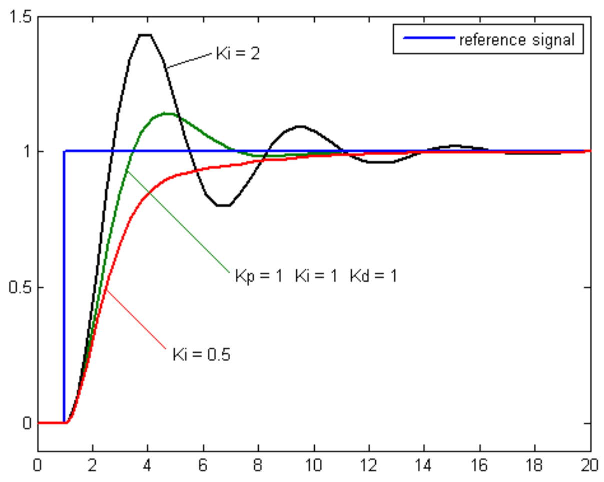
The derivative term of a PID controller represents the error's rate of change at time t, scaled by the derivative constant Kd. This is synonymous with the slope at time t, increasing when the slope of the error is very steep and decreasing with the slope is shallow.

Given that a PID controller is a sum of these three terms, its use should begin to make sense. As stated in the feedback example, PID controllers are filters that take the current and desired value of a system and output a value that can be applied to your system to properly move it to the desired position. Let us briefly examine a simple example of a PI controller to fully grasp its numerical inner workings.

Reusing the example of a simple robotic arm. We want to move the arm to an encoder position of 100 but the current encoder value is reading 30.

1. Prior to starting the program, we will first pick Kp and Ki gain values (Note that this process is typically done through trial and error). We will assign 0.1 to the Kp gain and 0.001 to the Ki term for now.
2. We will now begin the program and the first step is to determine the error by subtracting the current from the desired. In our case, this would be 100 minus 30, giving us an error of 70.
3. Taking the equation for the p-term, $K_p e(t)$, and replacing K_p for 0.1 and $e(t)$ for 70, the p-term comes out to equal (0.1×70), or 7.
4. Similarly, $K_i \int_0^t e(t)dt$ will be converted to $0.001 \int_0^1 70 * 20$. The value of dt depends on the sample rate of the program so I will assume this to be the conventional 20ms or 50hz. The answer comes out to 1.4
5. The p-term and i-term, 7 and 1.4 respectively, will now be added and produce the final output value of 8.4.

Depending on your method of motor control, this value could mean many things. However, this is a part of what makes PID controllers so useful: their versatility. If you were using a motor controller that takes percent-output values from 0 to 100, this value may be too low; in which case you might begin to tune the output by increasing the Kp gain.



A graph depicting various gains combinations and their resulting behavior

More values from the example:

t	current	p-term	i-term	output
1	30	7	1.4	8.4
2	35	6.5	2.7	9.2
3	47	5.3	3.76	9.06
4	65	3.5	4.46	7.96
5	78	2.2	4.9	7.1
6	96	0.4	4.98	5.38

The p-term decays rapidly near the setpoint leading to stall without a constantly accumulating i-term

Interpolation/Extrapolation

As you may have realized in the PID section, working with data sets are a very important skill for building effective robot code. Typically, data sets are used to tune values and debug logical errors in code; However, data sets are also used in some run-time evaluations as well. I.e. containers of values that are relevant to physical information about the robot or its tasks that are processed in real-time to evaluate instructions. A common example of a processed data set is a look-up table.

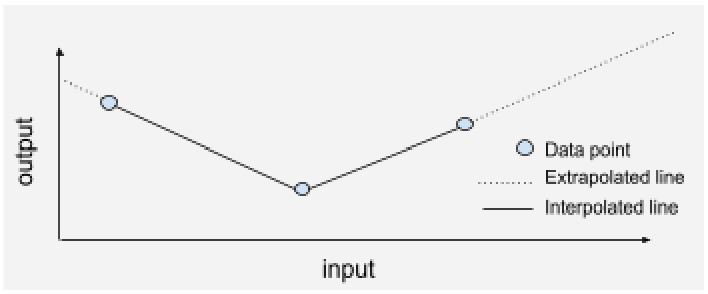
Look-up tables are tables that map values together. For example, take this graph of brightness to illuminate an object at various distances:

Distance from Target	Brightness of Light
10m	470 lm
30m	650 lm

One might query this table with its distance from the target and receive a brightness of light that it must produce to illuminate the target. This works fine if the target is always going to be either 10 or 30 meters away, but what about distances within or outside of these values? We will have to interpolate and extrapolate this data set. Interpolation is the approximation of values within the known values of a map, and extrapolation is the approximation of values outside the known values of a map.

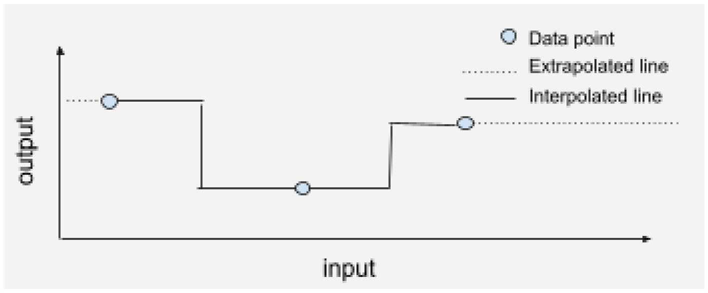
Interpolation and extrapolation come in various flavors. Some of the most frequently used include linear interpolation/extrapolation, nearest-neighbor interpolation/extrapolation, and spline interpolation/extrapolation.

Linear interpolation/extrapolation is the mathematical equivalent of drawing lines between know points and extending them at their ends. Interpolated values are simply taken as inputs to the linear equations defined by two points and extrapolated values are extensions of the linear equations at the endpoints.



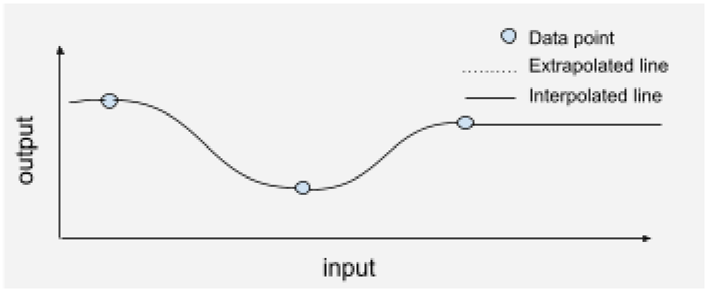
Do note that linear interpolation, and particularly extrapolation, are not typically used for values with sporadicity of this level. As the name implies, linear interpolations are used to express the relationship between linear systems. A linearly interpolated table would work particularly well with our distance and brightness example because only two values are provided and the relationship between brightness and distance is linear.

Nearest neighbor interpolation/extrapolation includes finding the nearest point to your input and outputting that point. Nearest neighbor interpolation is a form of interpolation/extrapolation that is not necessarily interpolation/extrapolation at all. Nearest neighbor methods are typically used in a setting where precision is unnecessary, the relationship is not linear, and the samples are abundant.



Disregarding the blatantly crooked lines, it is clear that the input values are simply passed to their "nearest neighbor" data points, and the output value of that point will be returned.

Spline interpolation/extrapolation is the most powerful, and also complex, of the three methods discussed. There are many, many types of spline methods but, at their core, they all consist of fitting non-linear mathematical functions over data points. If you are interested in learning the various spline interpolation methods, there are many helpful resources online. Some keywords include b-spline, Hermite spline, bezier splines, polynomial splines, and piecewise polynomial splines.



As mentioned, there are many different methods to interpolate data sets, and as such, the interpolating curves vary significantly in appearance and behavior: One major difference is the spline's end behavior. This particular spline flattens the curve towards the endpoints, and the extrapolated values will all lie on this extended line.

I mentioned that spline interpolation methods are powerful, but what makes them so? Despite their fancy appearance, systems rarely behave like explicit polynomials, so where are these utilized? One application of spline interpolation is trajectory planning.

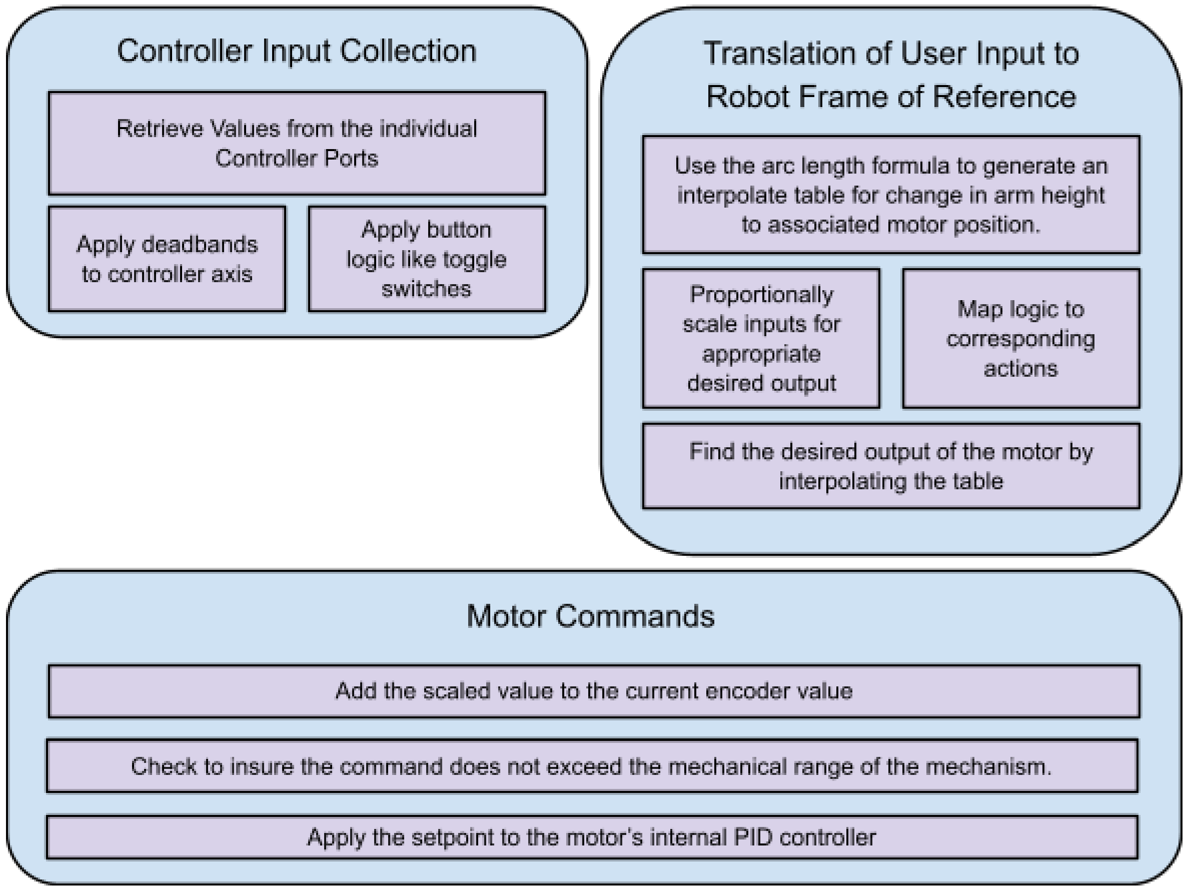
From vex, you may be familiar with orthogonal driving for robots. I.e. driving in a straight line in some direction for a specified amount. In FRC, with the help of wpilib api's, which will be covered in greater depth in the wpilib section, coordinate points can be passed in sequence and a robot will follow a smooth curve through all of the points. This functionality is far too advanced for this tutorial but you can learn more here. To explain briskly, the function takes data points and utilizes a Hermite clamped cubic or Hermite quintic spline method to interpolate the values and produce a smooth trajectory for the robot to follow.

Programming Robot Components

I would like to preface this section with some potentially insightful intuition. In writing robot code that can execute complex tasks, the complex task must first be translated into simple and sequential commands for the low-level actuators. It is a programmer's job, then, to separate tasks, allocate instruction, and organize resources. The case can be made that programming is all about organization: organization of state, organization of memory, and organization of functionality. The following few sections may seem trivial, excessively philosophical, or overtly pseudo-intellectual, but, In my experience, the most generally applicable ideas that I have learned relate to code organization or structure analysis.

What is Procedural Programming?

Let's say that a programmer wants to program a functional robot arm. The overall goal is to create a program that allows a user to control the robot arm with a controller. There are subsections to this larger task that may be split into controller input collection, translation of user input to robot frame of reference, and motor command. Going even deeper, these subsections have subsections of their own that may resemble the diagram below.


Controller Input Collection
Retrieve Values from the individual Controller Ports
Apply deadbands to controller axis
Apply button logic like toggle switches
Translation of User Input to Robot Frame of Reference
Use the arc length formula to generate an interpolate table for change in arm height to associated motor position.
Proportionally scale inputs for appropriate desired output
Map logic to corresponding actions
Find the desired output of the motor by interpolating the table
Motor Commands
Add the scaled value to the current encoder value
Check to insure the command does not exceed the mechanical range of the mechanism.
Apply the setpoint to the motor's internal PID controller

The breaking of large and complex tasks into smaller, less complex tasks is fundamental to programming in all contexts. How we decide to organize these tasks can, however, vary between design ideas called programming paradigms.

Procedural programming is a programming paradigm that may be called procedure-oriented programming as it centers around the functionality of the program. This means that the data and states are secondary to the functions and procedures. Within the underlying connection between program inputs and outputs, procedure calls operate on and alter state.

Most programming in Vex would fall under procedural. The input that is the controller values pass through conditions like dead-bands or functions that alter their values, and into the outputs for the motor commands.

Limitations of Procedural Programming

There are certainly ways to organize procedural programming in an intuitive and readable manner. However, I wish to make a clear-cut transition into the next topic so I will make little attempt to qualify. The limitation of procedural programming is its inability to protect state and organize attributes.

For example, a programmer might have a global variable within their code that can be read and written from anywhere in the program. This is a problem because it introduces the possibility of a race-case, or the unfacilitated altering of the variable's state in a way that disturbs how other procedures operate on it. We need a way to protect the variable by presenting an interface for protected access.

Additionally, take the example of a drive train. The drive train may consist of many motors, encoders, or gyros in any combination. Say that we simply wanted to move the drive train 500 revolutions forward. We would have to go to every motor and set each of their setpoints to 500 manually. If there were a way for us to easily interact with the drive base instead of all of its internal parts— to wrap the complexity in a neat box— that may increase its readability as well as its safety.

Intro to OOP/Object-Oriented Programming

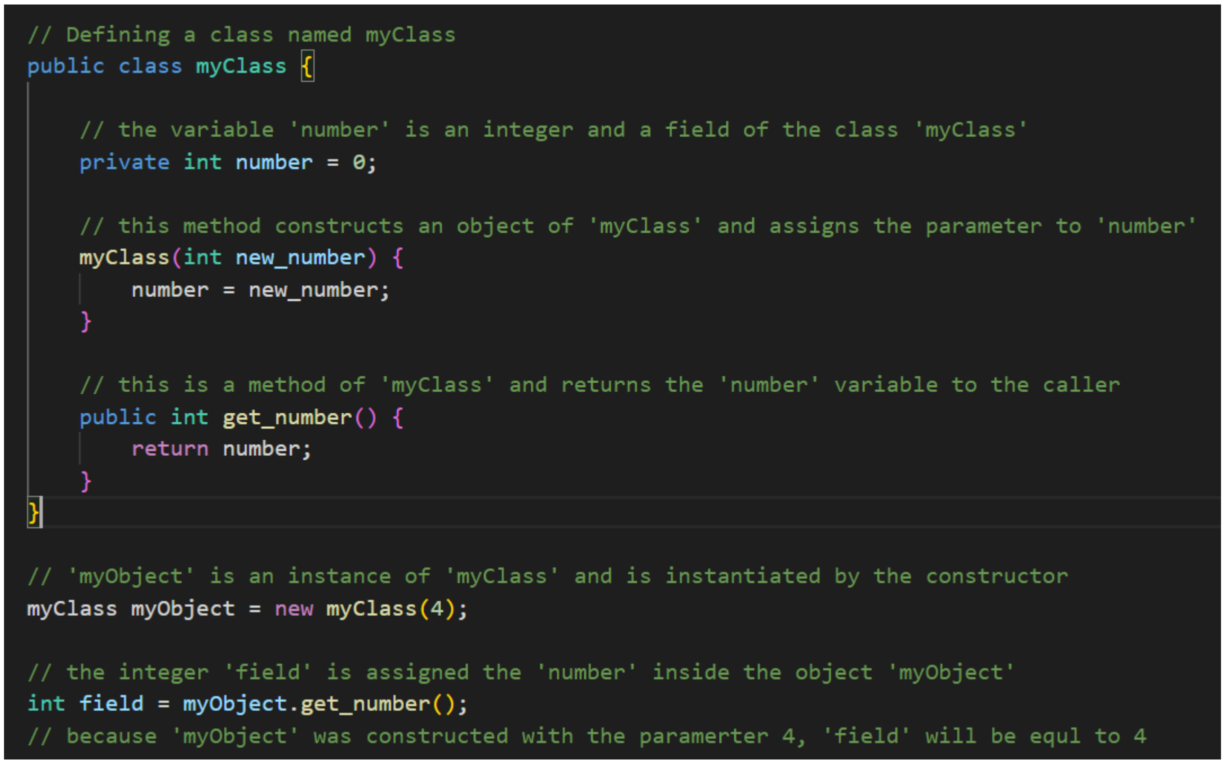
Philosophy

Object-oriented programming is a programming paradigm that centers around the characterization of states and data types with a secondary emphasis on the procedures and functions that act on them. The OOP paradigm was created to encapsulate data and states with relational significance into a single entity.

In order to achieve this data organization, the OOP paradigm introduces a new data structure called classes. You can think of classes like other data types such as int (integer), float (decimal), and bool (boolean), but classes have special characteristics that make them quite different from their simpler counterparts.

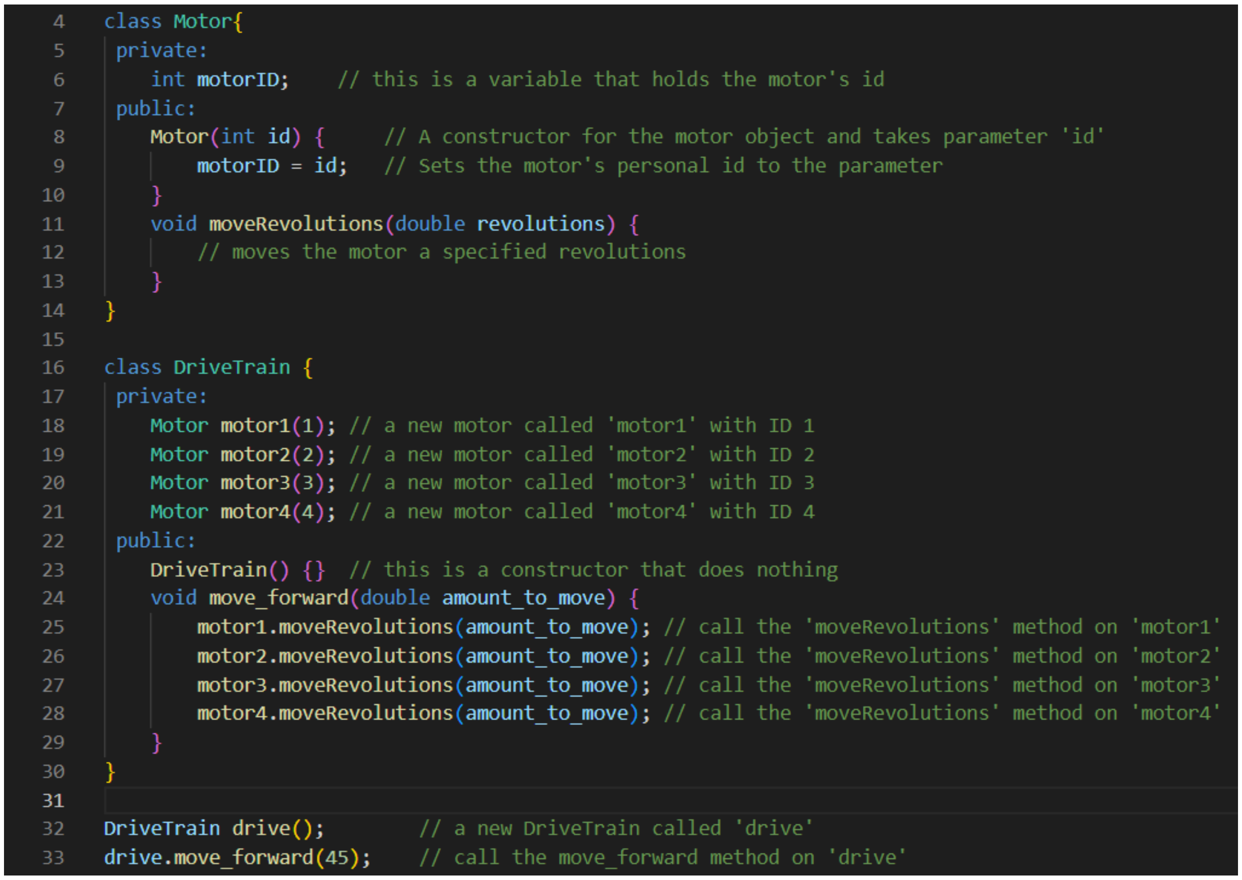
- Classes consist of fields and methods
- Fields are variables that belong to the class
- Methods are functions that belong to the class

A class is like a blueprint that includes methods and fields. When you instantiate a class, i.e. make an 'object' of the class, the compiler has to look at this blueprint to create it. However, once it is created, the new object is entirely independent of its class. The new object initially stores the same methods and fields as the class from which it was birth, but the fields can then be altered and their changes will not propagate to the class. Additionally, you can make any number of objects from one class. Here is a snippet of code from Java, a language that will be discussed in the Java section:



```java
// Defining a class named myClass
public class myClass {

    // the variable 'number' is an integer and a field of the class 'myClass'
    private int number = 0;

    // this method constructs an object of 'myClass' and assigns the parameter to 'number'
    myClass(int new_number) {
        number = new_number;
    }

    // this is a method of 'myClass' and returns the 'number' variable to the caller
    public int get_number() {
        return number;
    }
}

// 'myObject' is an instance of 'myClass' and is instantiated by the constructor
myClass myObject = new myClass(4);

// the integer 'field' is assigned the 'number' inside the object 'myObject'
int field = myObject.get_number();
// because 'myObject' was constructed with the paramerter 4, 'field' will be equl to 4
```

I'll now respond to the question raised in the Limitations of Procedural Programming by addressing how an object-oriented program might approach the characterization of a drive train. The drive train will be represented as its own class consisting of 4 motors. Each motor is an object of a motor class (this class is often provided by the platform we are working for). When the program executes, we should make one object of "DriveTrain" and access the methods to command the entire drive train. Below is a snippet of pseudo-C++ that demonstrates this concept.



```cpp
class Motor{
 private:
    int motorID;    // this is a variable that holds the motor's id
 public:
    Motor(int id) {     // A constructor for the motor object and takes parameter 'id'
        motorID = id;   // Sets the motor's personal id to the parameter
    }
    void moveRevolutions(double revolutions) {
        // moves the motor a specified revolutions
    }
}

class DriveTrain {
 private:
    Motor motor1(1); // a new motor called 'motor1' with ID 1
    Motor motor2(2); // a new motor called 'motor2' with ID 2
    Motor motor3(3); // a new motor called 'motor3' with ID 3
    Motor motor4(4); // a new motor called 'motor4' with ID 4
 public:
    DriveTrain() {}  // this is a constructor that does nothing
    void move_forward(double amount_to_move) {
        motor1.moveRevolutions(amount_to_move); // call the 'moveRevolutions' method on 'motor1'
        motor2.moveRevolutions(amount_to_move); // call the 'moveRevolutions' method on 'motor2'
        motor3.moveRevolutions(amount_to_move); // call the 'moveRevolutions' method on 'motor3'
        motor4.moveRevolutions(amount_to_move); // call the 'moveRevolutions' method on 'motor4'
    }
}

DriveTrain drive();         // a new DriveTrain called 'drive'
drive.move_forward(45);     // call the move_forward method on 'drive'
```

Notice how we created a blueprint for motors and a blueprint for the drive train with classes. In the entire code, the only 'executed' pieces of code are the bottom two lines where we create an object of the class 'DriveTrain' called 'drive', and call the 'move_forward' method. Note, however, that when we construct the 'drive' object, four objects of motors are automatically instantiated as fields of the 'DriveTrain' class.

Encapsulation

Encapsulation is the act of encapsulating data and states into classes as well as protecting states through classes. As mentioned earlier, the primary reason for OOP's conception was the recurring computer science problem of shared state. OOP offers a solution to the shared state problem by discretizing states and forging communication between states through getters and setters. Essentially, OOP forces programs to not share the states themselves, but only references to them.

You may have noticed the occasional appearance of the words public and private in the examples above. For most OOP languages, when encapsulating fields or protecting them from outside influence, the private keyword is used. This means that only methods of the same class have access to the private fields, and attempts to directly see or change the state from the outside are denied.

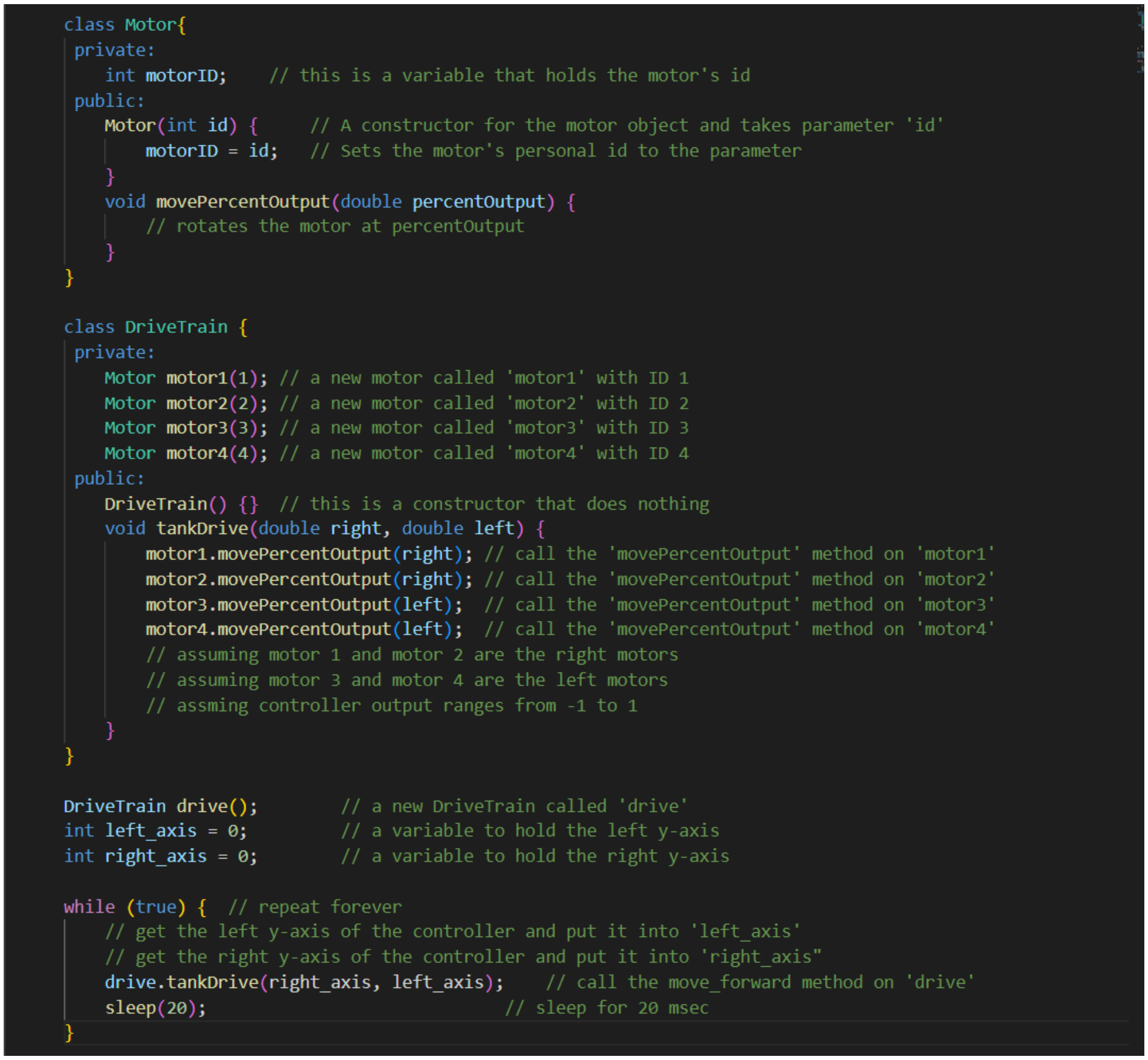
Therefore, dedicated methods called getters and setters — used to query a field's state and alter a field's state, respectively — are needed. The advantage of this seemingly roundabout approach is the layer of filtering that can be placed within the getters and setters. For example, a filter that ensures incoming values are within an allowable range could be placed in the setter for a private variable, thereby protecting its state from unexpected behavior. There are, of course, many other ways to protect states in various cases such as only allowing the altering of a temporary variable that conditionally interacts with the 'real' private state, but the point should be clear by now:

Encapsulation is both a means of organizing data into larger entities and protecting states from unauthorized access.

Abstraction

Abstraction is the idea of hiding complex tasks behind a simple interface. Much like a function's job to hold reusable code in a package that is easy to use, abstraction in OOP is the packaging of code into classes. In the drivetrain example, we wished to 'abstract' the task of sending commands to all the motors, so we created a method that simply took one number and applied it to all of the motors. Though this is a good example, abstraction can go much further in simplifying the programmer's job outside of the classes.

Suppose that we want to make a tank-drive operator-control code. In the drive train class, we could define a method that takes the y-axis values of the left and right joystick and move the robot accordingly. We might make an alteration like this to the drive train program:

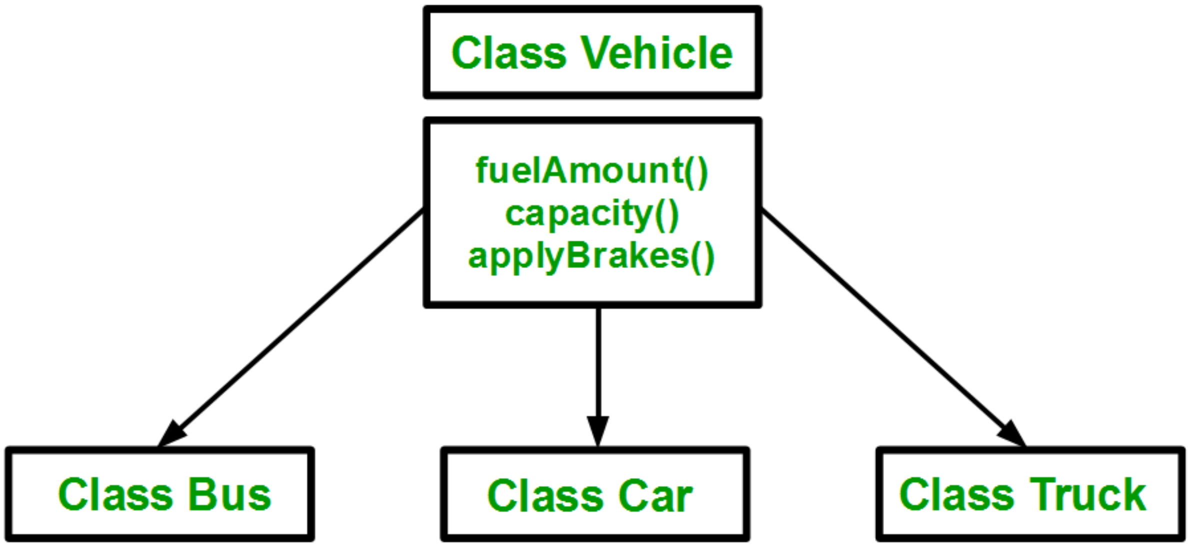


```
class Motor{
 private:
    int motorID;    // this is a variable that holds the motor's id
 public:
    Motor(int id) {     // A constructor for the motor object and takes parameter 'id'
        motorID = id;   // Sets the motor's personal id to the parameter
    }
    void movePercentOutput(double percentOutput) {
        // rotates the motor at percentOutput
    }
}

class DriveTrain {
 private:
    Motor motor1(1); // a new motor called 'motor1' with ID 1
    Motor motor2(2); // a new motor called 'motor2' with ID 2
    Motor motor3(3); // a new motor called 'motor3' with ID 3
    Motor motor4(4); // a new motor called 'motor4' with ID 4
 public:
    DriveTrain() {}  // this is a constructor that does nothing
    void tankDrive(double right, double left) {
        motor1.movePercentOutput(right); // call the 'movePercentOutput' method on 'motor1'
        motor2.movePercentOutput(right); // call the 'movePercentOutput' method on 'motor2'
        motor3.movePercentOutput(left);  // call the 'movePercentOutput' method on 'motor3'
        motor4.movePercentOutput(left);  // call the 'movePercentOutput' method on 'motor4'
        // assuming motor 1 and motor 2 are the right motors
        // assuming motor 3 and motor 4 are the left motors
        // assming controller output ranges from -1 to 1
    }
}

DriveTrain drive();        // a new DriveTrain called 'drive'
int left_axis = 0;         // a variable to hold the left y-axis
int right_axis = 0;        // a variable to hold the right y-axis

while (true) {  // repeat forever
    // get the left y-axis of the controller and put it into 'left_axis'
    // get the right y-axis of the controller and put it into 'right_axis"
    drive.tankDrive(right_axis, left_axis);    // call the move_forward method on 'drive'
    sleep(20);                             // sleep for 20 msec
}
```

The tank drive function is now seamlessly integrated into the drive train class and the programmer merely calls the 'tankDrive' method. Warning: writing commands in mechanism classes is considered bad practice in the wpilib command base framework

Polymorphism

Polymorphism, in the context of OOP, is the hierarchy of descendants between classes. In many languages that support OOP, classes can perform inheritance. I.e. they can inherent fields and methods from parent classes. A common illustration can be seen in the following graphic:



A car, bus, and truck class inherent methods from a vehicle class. - Via GeeksforGeeks

You could say that the vehicle class is polymorphic because it appears in multiple (poly) forms (morphic) as a truck, class, and bus. However, because polymorphism typically indicates larger trees of inheritance that are simply non-existent in FRC programs, I will shift the primary focus to inheritance.

In OOP, inheritance is the downwards propagation of attributes, i.e. fields and methods, through classes with shared relationships. In the example above, the classes bus, car, and truck are children of the vehicle class because they are specific versions of a vehicle (A bus is a vehicle but a vehicle is not always a bus). The bus, car, and truck all inherit the methods of vehicle class.

Typically, inheritance is used to reuse code amongst things that are related in a categorical manner. In FRC, inheritance is generally only used to give programmers a framework to develop specific structures such as subsystems and commands. This concept will be expanded upon in the java overview, but user-defined subsystems and commands inherent their base classes in order to possess important methods required for their framework.

WPILib

Introduction

According to the wpilib docs,

> “ The WPI Robotics Library (WPILib) is the standard software library provided for teams to write code for their FRC® robots. A software library is a collection of code that can be imported into and used by other software. WPILib contains a set of useful classes and subroutines for interfacing with various parts of the FRC control system (such as sensors, motor controllers, and the driver station), as well as an assortment of other utility functions. “

To put it another way, the WPI library, or wpilib for short, is a collection of features and frameworks on which to build your code.

WPILIB is an open source project that strives to make programming an FRC robot easier for newer teams. The library itself contains generic classes and functions that are frequently used in frc, and a simple import of the library will allow teams to access all of its functionality.

Documentation

The wpilib documentation can be found here, and details the entire process from the software setup in their zero to robot, to the software use in the API’s. The page itself is great introduction to frc software and much of what is not covered here can be found there. I highly recommend you at-least skim over the entire page.

Java

Introduction

Java is a programming language developed by Sun Microsystems in the '90s as a high-level object-oriented extension of c and c++ with "write once run anywhere" functionality. Java, unlike most languages, only supports objected-oriented programming. This means everything from the main procedure to the global procedures and the constants container, are classes.

Though this introduction will refrain from explaining the technical nooks and crannies of the Java language, I will briefly explain the important parts of access and non-access modifiers.

Access Modifiers

Access modifiers are keywords that modify the access permissions of methods, classes, and other data types. There are several access modifiers in Java, but the main two, private and public, are frequently used in FRC programming.

Private: When used on data types such as variables and classes, private modifiers modify the permission of the element to be only accessible within the class in which it is declared. This means that a private variable within a class can only be read and written by a method within the same class, thus enforcing the OOP concept of encapsulation.

Public: When used on data types such as variables and classes, public modifiers modify the permission of the element to be accessible from anywhere in the program. Variables and classes that are needed in various parts of the program can be modified to be public. As covered in the encapsulation section, be especially careful when publicizing data types that are susceptible to erroneous manipulation. Specifically, variables should almost always be of private access permission.

Non-Access Modifiers

Non-access modifiers are keywords that alter the behavior of the element or data type on which it is used. There are even more non-access modifiers, but the main two for use in FRC are static and final.

Static: The static keyword can only be used on attributes and methods and modify the ownership of the element to the class rather than the individual objects of the class. Static variables, for instance, propagate their state to the classes so all instances of the class share the same value within the variable.

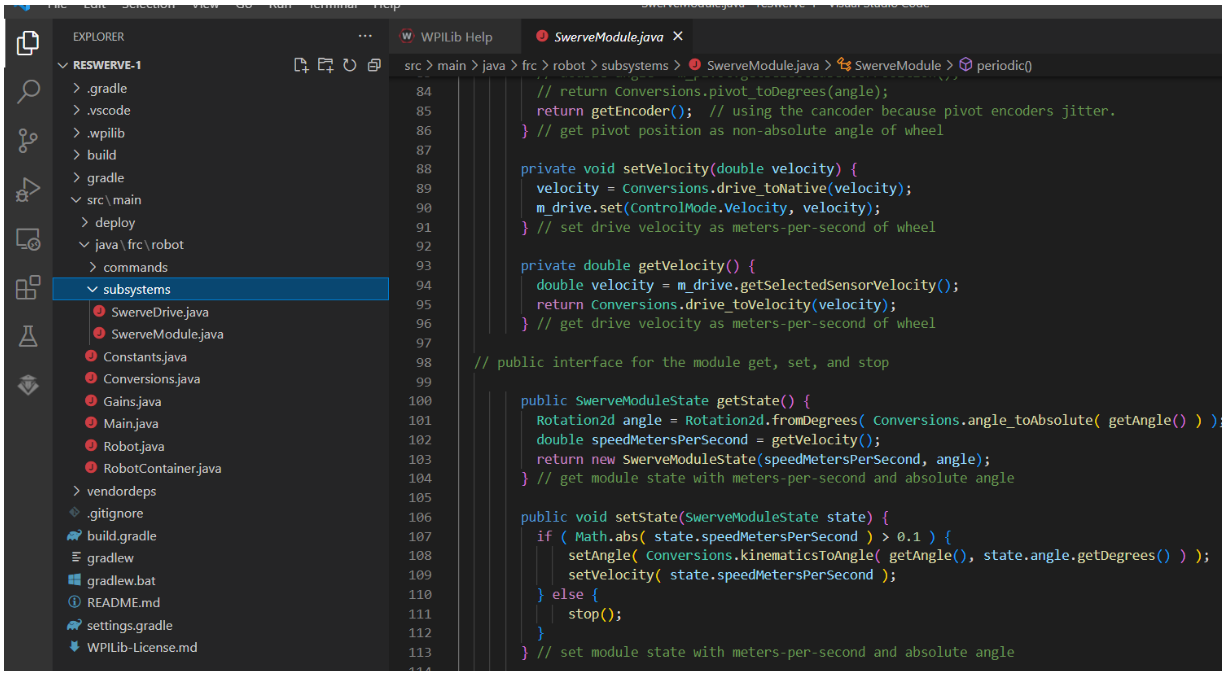
Final: When used on classes, the final keyword simply ensures that the class is not inheritable. When used on methods, the final keyword ensures it can not be overridden by subclasses. When used on a variable, the variable is only allowed to be initialized once in run-time.

The Command Robot Structure

The command robot structure is one of two main programming structures in the wpilib framework. The command structure divides structure and procedure into subsystems and commands which are called elsewhere in the program. These classifications, like much OOP design, are trivial and merely organizational, but when working in large groups as you would do in an FRC team, these trivial organizations can go a long way in standardizing code construction.

Subsystems

Subsystems are classes that store the behavior for a corresponding physical component. Found within the “subsystems” folder of the robot project, a single file contains a single public class.



```java
      // return Conversions.pivot_toDegrees(angle);
      return getEncoder();  // using the cancoder because pivot encoders jitter.
    } // get pivot position as non-absolute angle of wheel

    private void setVelocity(double velocity) {
      velocity = Conversions.drive_toNative(velocity);
      m_drive.set(ControlMode.Velocity, velocity);
    } // set drive velocity as meters-per-second of wheel

    private double getVelocity() {
      double velocity = m_drive.getSelectedSensorVelocity();
      return Conversions.drive_toVelocity(velocity);
    } // get drive velocity as meters-per-second of wheel

  // public interface for the module get, set, and stop

    public SwerveModuleState getState() {
      Rotation2d angle = Rotation2d.fromDegrees( Conversions.angle_toAbsolute( getAngle() ) );
      double speedMetersPerSecond = getVelocity();
      return new SwerveModuleState(speedMetersPerSecond, angle);
    } // get module state with meters-per-second and absolute angle

    public void setState(SwerveModuleState state) {
      if ( Math.abs( state.speedMetersPerSecond ) > 0.1 ) {
          setAngle( Conversions.kinematicsToAngle( getAngle(), state.angle.getDegrees() ) );
          setVelocity( state.speedMetersPerSecond );
      } else {
          stop();
      }
    } // set module state with meters-per-second and absolute angle
```

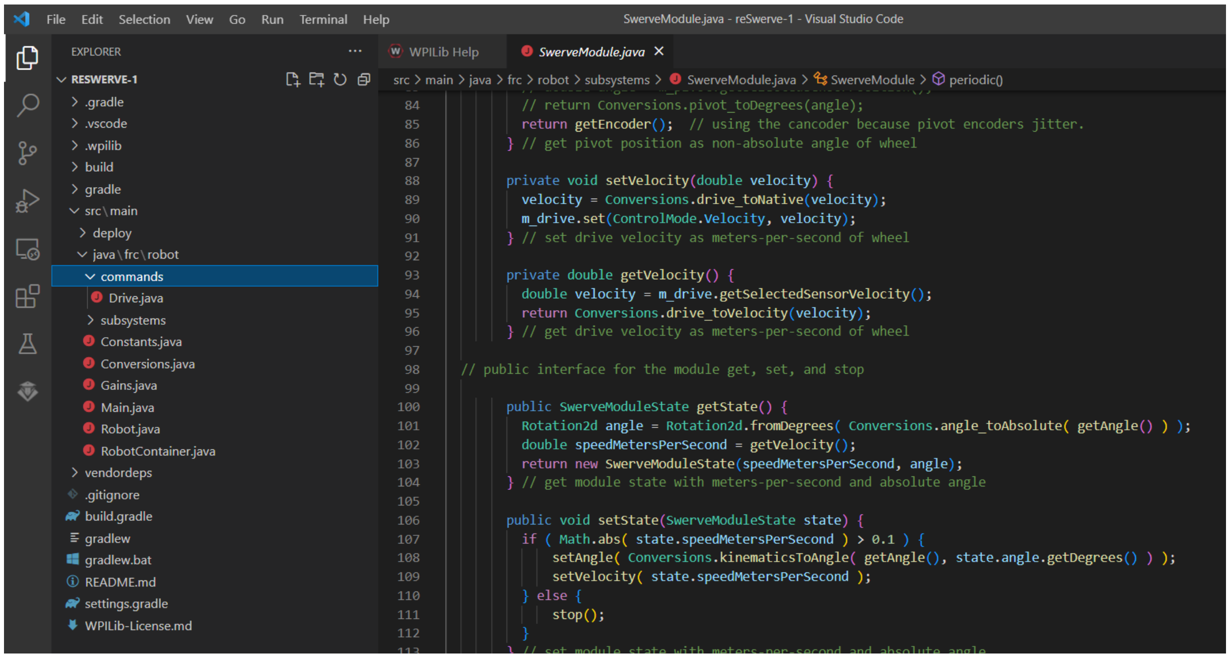
A “SwerveDrive” and “SwerveModule” file within the “subsystems” folder.

Subsystems are, in short, components of the robot that physically interact together or share components and working area. It is quite difficult to put an exact and universal definition on subsystems— even determining the subsystems that exist within your robot can be quite challenging. Subsystems are a means of distinguishing and separating unrelated portions of a robot code such as the drivetrain and the shooter; However, the distinction between information relation and physical relation, that is, the difference between components that frequently share information and components that share the same physical components, can significantly blur the lines between their differentiation. Typically, the physical interpretation of subsystems distinguish their separation within subsystems and the information they may exchange is facilitated by interactions in code. Of course, there is no correct answer, but I digress,

Subsystem classes typically contain private instances of the motors and actuators that are a part of the physical component, and public methods within the subsystem create an interface for other portions of the code to access them. This abstraction of the low-level components that make up the larger subsystem is the primary task of the subsystem in the command based framework.

Commands

As is indicated in the name, the existence and use of commands define the command-based structure of code. Commands are individual classes, within individual files, within the “commands” folder, and store the code for a specific behavior of a subsystem.



```java
      // return Conversions.pivot_toDegrees(angle);
      return getEncoder();  // using the cancoder because pivot encoders jitter.
    } // get pivot position as non-absolute angle of wheel

    private void setVelocity(double velocity) {
      velocity = Conversions.drive_toNative(velocity);
      m_drive.set(ControlMode.Velocity, velocity);
    } // set drive velocity as meters-per-second of wheel

    private double getVelocity() {
      double velocity = m_drive.getSelectedSensorVelocity();
      return Conversions.drive_toVelocity(velocity);
    } // get drive velocity as meters-per-second of wheel

  // public interface for the module get, set, and stop

    public SwerveModuleState getState() {
      Rotation2d angle = Rotation2d.fromDegrees( Conversions.angle_toAbsolute( getAngle() ) );
      double speedMetersPerSecond = getVelocity();
      return new SwerveModuleState(speedMetersPerSecond, angle);
    } // get module state with meters-per-second and absolute angle

    public void setState(SwerveModuleState state) {
      if ( Math.abs( state.speedMetersPerSecond ) > 0.1 ) {
          setAngle( Conversions.kinematicsToAngle( getAngle(), state.angle.getDegrees() ) );
          setVelocity( state.speedMetersPerSecond );
      } else {
          stop();
      }
    } // set module state with meters-per-second and absolute angle
```

A “Drive” file within the “commands” folder.

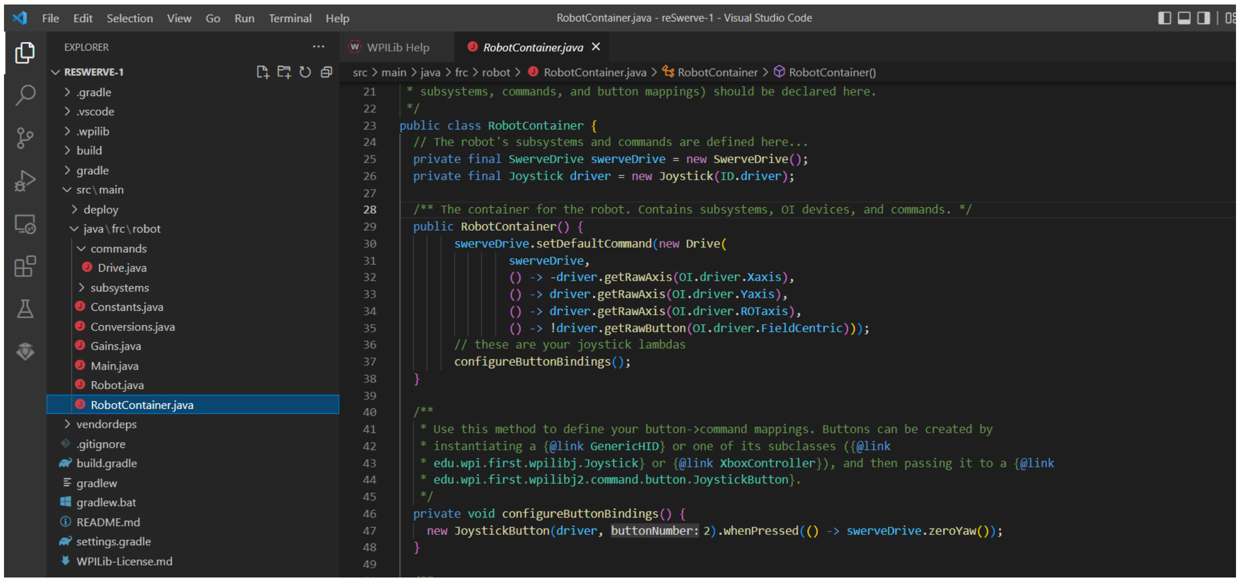
Although it may not seem so at first, structuring robot code as combinations of commands in sequence differs quite significantly from typical procedural programming. Commands work to discretize the instructions for the subsystems as a certain, strictly defined, function; as opposed to the procedural programming method of classifying instructions as a collage of low-level instruction. One benefit of “command-ifying” instructions for the subsystem is the programmer’s ability to easily tell when a program is running a specific piece of instruction, as well as what that particular instruction entails.

Another method of discretizing instructions is through the use of subsystem state manipulation. In this structure, subsystems hold a list of states that they could possibly be in, and outside functions operate only on the variable holding the state while the subsystem is tasked with executing the specific instruction for the specified state. This, however, is not utilized in the command structure.

Commands are disposable instances of their class. This means that they are created to initiate the command and subsequently destroyed after termination by the Java garbage collection system. The command classes themselves tend to hold high-level instructions that interface with one or multiple subsystems such as driver control for the drive base or target tracking using a combination of the drive base and turret.

Robot Container

The robot container is a class representing the robot and all of its functionality. When the robot container class is initialized, objects of the subsystems are finally created and commands are mapped to various schedulers.



RobotContainer.java - reSwerve-1 - Visual Studio Code
EXPLORER
RESWERVE-1
WPILib Help
RobotContainer.java
src > main > java > frc > robot > RobotContainer.java > RobotContainer > RobotContainer()

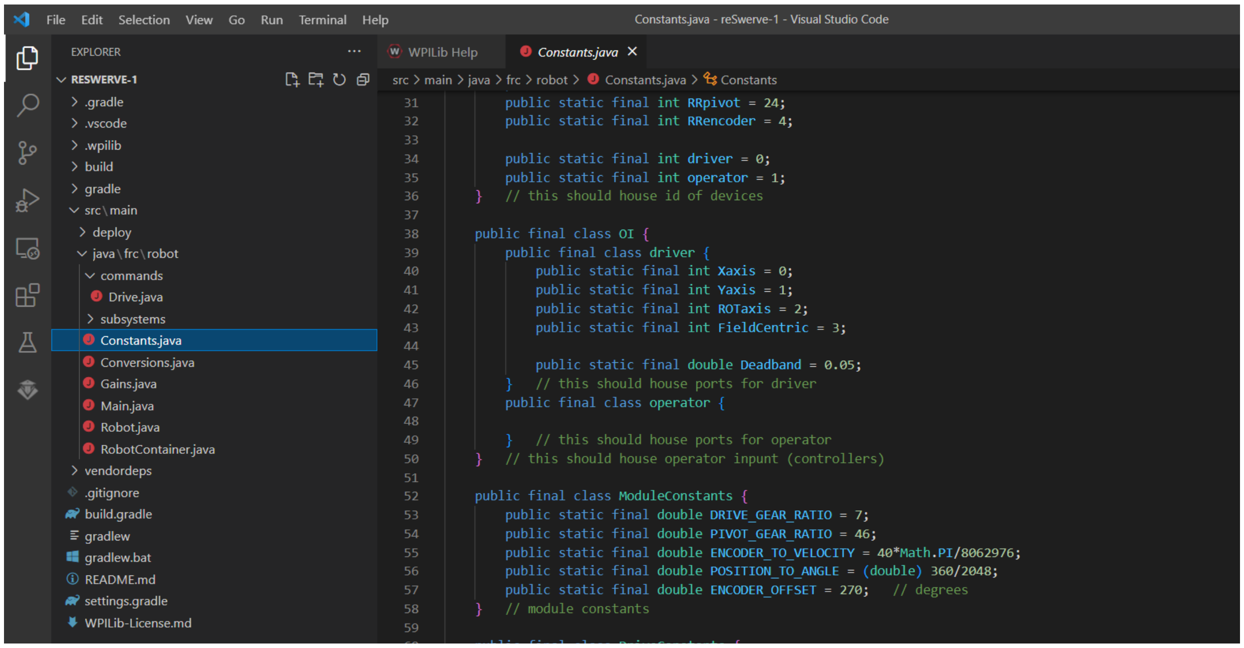
```java
 * subsystems, commands, and button mappings) should be declared here.
 */
public class RobotContainer {
  // The robot's subsystems and commands are defined here...
  private final SwerveDrive swerveDrive = new SwerveDrive();
  private final Joystick driver = new Joystick(ID.driver);

  /** The container for the robot. Contains subsystems, OI devices, and commands. */
  public RobotContainer() {
        swerveDrive.setDefaultCommand(new Drive(
                swerveDrive,
                () -> -driver.getRawAxis(OI.driver.Xaxis),
                () -> driver.getRawAxis(OI.driver.Yaxis),
                () -> driver.getRawAxis(OI.driver.ROTaxis),
                () -> !driver.getRawButton(OI.driver.FieldCentric)));
        // these are your joystick lambdas
        configureButtonBindings();
  }

  /**
   * Use this method to define your button->command mappings. Buttons can be created by
   * instantiating a {@link GenericHID} or one of its subclasses ({@link
   * edu.wpi.first.wpilibj.Joystick} or {@link XboxController}), and then passing it to a {@link
   * edu.wpi.first.wpilibj2.command.button.JoystickButton}.
   */
  private void configureButtonBindings() {
    new JoystickButton(driver, buttonNumber: 2).whenPressed(() -> swerveDrive.zeroYaw());
  }
```

A “RobotContainer” file within the “robot” folder.

As stated above, the robot container initializes all of the subsystems, controllers, and other miscellaneous non-user-defined device classes. The robot container also configures operator inputs to commands and stores autonomous routines.

Constants

The constants file/class includes all of your code's variable constants. I.e. variables that do not change in value. Some example of a constant variable is the circumference of the drive wheel, used to calculate the movement of the robot for a unit rotation of the drive motor.



```java
        public static final int RRpivot = 24;
        public static final int RRencoder = 4;

        public static final int driver = 0;
        public static final int operator = 1;
    }   // this should house id of devices

    public final class OI {
        public final class driver {
            public static final int Xaxis = 0;
            public static final int Yaxis = 1;
            public static final int ROTaxis = 2;
            public static final int FieldCentric = 3;

            public static final double Deadband = 0.05;
        }   // this should house ports for driver
        public final class operator {

        }   // this should house ports for operator
    }   // this should house operator input (controllers)

    public final class ModuleConstants {
        public static final double DRIVE_GEAR_RATIO = 7;
        public static final double PIVOT_GEAR_RATIO = 46;
        public static final double ENCODER_TO_VELOCITY = 40*Math.PI/8062976;
        public static final double POSITION_TO_ANGLE = (double) 360/2048;
        public static final double ENCODER_OFFSET = 270;    // degrees
    }   // module constants
```

A "Constants" file within the "robot" folder.

Unchanging and universal values are often kept in the constants class because they are:

1. accessed in multiple parts of the programm, requiring global properties, and
2. Allow less error-prone updating of variables to reflect physical changes

For instance, if the diameter of the drive wheel were to change from 4 inches to 5 inches, the circumference of the wheel would, of course, change as well. Without having to search for the location of the circumference variable in some distant and unintuitive area of the code, we could simply look within the constants and change the value there.

Additionally, it is typically, and even encouraged, that further classes within the constants class are devised for organizing the variables. In the picture above, the "OI" class and the "ModuleConstants" class are subclasses of the "Constants" class, and create a distinction between the controller and module constants. To access the values within these variables, one would simply import the constants file and write the class path with the dot operator. For example, accessing the "Deadband" variable is as simple as calling "Constants.OI.Deadband", or just "OI.Deadband" if you imported the OI class directly.

Note that an instance of the controller class is unnecessary to access the constant variables because the variables are all of the static type. I.e. they belong to the class and not the object so they can be called without creating an object of the class.

The Calling Path

When I was studying the topic mere months ago, I found the lack of documentation on the exact calling path of the program frustrating. I felt that understanding the exact order of execution for the program would significantly increase my ability to understand the structure. With that, albeit slightly speculatory, here is a briefing of the calling and initialization path of the program. Hopefully, this knowledge will help significantly in approaching example programs on your own.

1. Like any Java program, the program begins execution with the “main” class within the “Main” file. The main class always follows the “public static void main (string args)” format, and in the case of the command base framework, passes a new object of “Robot” to the framework application runner.

2. The “Robot” class within the “Robot” file contains functions that are representations of the current state of the program including periodic execution, teleoperated period, and autonomous period, among others. The framework application runner will call these functions according to the state of FMS (field management system) or the driver station. Of course, there may be user defined commands and objectives within some of these functions, but within the raw template, there are three notable calls. Firstly, inside the “robotInit” function, the robot container (recall this contains all of the robot components and functionality) is initialized. Next, within the “robotPeriodic” function, the command scheduler, a tool for running commands, is instructed to run. Finally, in the “autonomousInit” function, the executed autonomous routine is fetched from the robot container.

Now, when the program is initially started, “Robot” will be initialized and the initialization for the robot container will commence. When the robot container is created, the code no longer cares for the individual subsystems and commands because they are all initialized and mapped to specific triggers within the robot container. When commands are triggered within the robot container, they are scheduled to the command scheduler, which, if you recall, is called to be executed every cycle by the periodic executor.

Source Control

Intro to Source Control

When working as a team to write code, a distinguished and safe method of file sharing is necessary to combine the various works of various people into a single project. This is called source control in programming and an important skill for a team programmer and those that wish to pursue programming after high school.

The conventional method of source control deals with remote and local repositories; Repositories are like virtual folders that contain code and other associated files.

Remote repositories are repositories that are uploaded to the cloud. In order to make a repository accessible to a team member on any device, they are initially created on a remote server. Think of the remote repository as the main folder that is immune to hardware failures such as hard drive wipes and accessible to anyone, anywhere (that has been permitted to do so).

Local repositories are repositories that exist on local devices. When a team member wants to make a change to the code, they will first have to make a copy of the remote repository on their local device. After making changes to their local repository, the team member will proceed to upload their local repository to the remote repository so that every other team member has access to the latest version.

VS code, Git, and GitHub

"*[Github] is a provider of Internet hosting for software development and version control using Git* **(wikipedia)**" Basically, Github is a cloud service like the google suit, but for software developers. Similar to a google drive, where users can upload files to the internet and access them through their account on any device, GitHub offers cloud services on which users can upload files and access them remotely. However, unlike Google drive, GitHub contains added functionality that is specific to software development, namely Git, that you will surely come to observe by spending some time with the application.

VS code is a software called an IDE, or integrated development environment. At its core, VS code allows users to open and edit code on a user-friendly and fast interface. There are, however, many IDEs to choose from, and what makes VS code special is the additional functionality that separates it from its competitors. Intellisense, for instance, a tool that includes features such as context-aware auto-completion, or syntax error detection, is particularly useful in VS code. VS Code also offers built-in tools to access Git, and with it, GitHub.

"*Git is software for tracking changes in any set of files, usually used for coordinating work among programmers collaboratively developing source code during software development* **(Wikipedia)**." So, Git is a version control software that allows developers to seamlessly collaborate on code. Git could be considered a tool that can function as the bridge between the developers on individual code editors such as VS Code, and cloud services such as GitHub.

You will continue to learn the specifics of these applications, their use-cases, and their operation instructions, but for now, that marks the end of this tutorial.