Learn Java for FTC

Alan G. Smith

May 7, 2020

Learn Java for FTC

The author can be contacted at: alan@ftcteams.com
The hardcopy of the book can be purchased from Amazon
The most recent PDF is free at https://github.com/alan412/LearnJavaForFTC

ISBN: 9798644009886

This book is dedicated to:

My wife who after suffering through my first book encouraged me to write this one.

My FTC team that excites me about teaching

My father who spent many hours with me on the Vic 20, Commodore 64, and the robotic arm science project. Without his investment, I wouldn't be the engineer I am today.

All who would desire to teach students how to think and work hard.

Whatever you do, work at it with all your heart, as working for the Lord, not for men.
Colossians 3:23 (NIV 1984)

Contents

1. Introduction

In coaching an FTC team[1], I found that students wanted to be effective coders but had trouble figuring out where to start. When they took online courses, they ended up learning a lot of things that weren't helpful for FTC. (or even usable) In addition, many of the online sources and even books teach bad habits. I started this as some slides for my team, but decided it would be better as a book that could be shared widely.

1.1. Hardware

1.1.1. Robot Controller

The robot controller is either an Android Phone or a REV Control Hub. It is the "brains" that are actually on your robot. We often abbrievate Robot Controller as "RC". A Rev Control Hub is new for the 2020-2021 FTC Season and is basically an Android Phone inside of an Expansion Hub so that instead of a Robot Controller phone connected to a Expansion Hub, a Control Hub is both of those parts in the same pacakge.

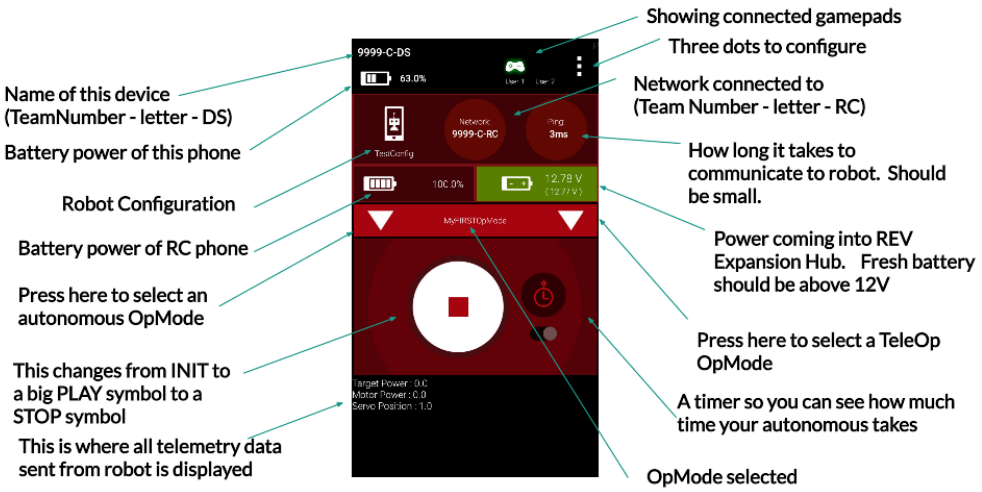
1.1.2. Programming Board

For this book, we have made a simple *Programming Board* that we can use throughout the book so that we all have the same hardware. For directions on how to make your own, see Appendix A

In addition, there is the driver station (a phone with 1 or 2 USB gamepads connected)., and a robot controller. The robot controller can be either a phone or a REV control Hub.

[1] Go Quantum Quacks - FTC #16072

1.1.3. Driver Station



The driver station is an Android Phone with USB gamepads connected that are used during the game to drive the robot. Above is an example driver station with descriptions for everything on it. This changes some from year to year. We often abbreviate the Driver Station as DS.

1.2. Our first OpMode

1.2.1. What is an OpMode?

A little terminology before we get started.

class In Java all code is grouped together in classes. We'll discuss exactly what classes are later in chapter 5. For now, just know that a class groups like code together and in Java, each class is in its own file that is named the same as the class with `.java` at the end.

method A class can have methods which are code that is the smallest group of code that can be executed. It is like a function in some languages or a MyBlock in EV3-G. We'll talk more about this later in section 5.2.

package A directory in JAVA. Files in the same package have special privileges with each other. And yes, a package can have packages within it.

In FTC, An OpMode is a program for our robot. We can have multiple OpModes. They are all stored in the TeamCode package.

1.2.2. Parts of an OpMode

In FTC, An OpMode[2] is a program for our robot.

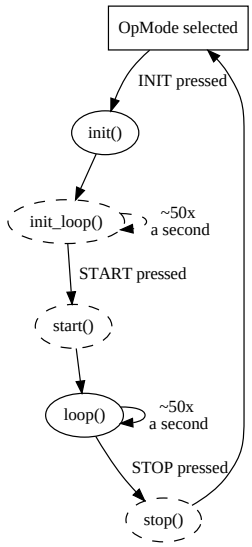
[2]You may run across LinearOpMode. There is a discussion in Appendix B for why we don't use it but it is probably best left for the end.

OpModes are required to have two methods:

1. `init()` - This is run **once** when the driver presses INIT.
2. `loop()` - This is run **repeatedly** after driver presses PLAY but before STOP.

In addition, there are three optional methods. These are less common but can be very useful.

1. `init_loop()` - This is run **repeatedly** after driver presses INIT but before PLAY.
2. `start()` - This is run **once** when the driver presses PLAY.
3. `stop()` - This is run **once** when the driver presses STOP.



If you look over on the right, you'll see a diagram that explains roughly how it works. The solid ovals are required and the dashed ones are optional. After `stop()` is executed it goes back to the top.

I know this seems like a lot of strangeness, but I promise it will make more sense as we continue.

1.2.3. Hello, World

Traditionally, the first program written in every programming language simply writes "Hello, World!" to the screen. But instead of writing to the robot's screen, we'll write to the screen on the Driver Station. (Throughout this book we will show the program in its entirety first, and then explain it afterwards. So if you see something that doesn't make sense, keep reading and hopefully it will be cleared up.)

Listing 1.1: `HelloWorld.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class HelloWorld extends OpMode {
8      @Override
9      public void init() {
10         telemetry.addData("Hello","World");
11     }
12
13     @Override
14     public void loop() {
15
16     }
```

```
17 }
```

Here is a breakdown of what this program does.

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
```

If you are working in Android Studio, you won't have to enter any of these lines as it will add them for you. Line 1 basically says where this file is located. 3 and 4 bring in code from the FTC SDK (Software Development Kit) so we can use them.

```
6  @TeleOp()
```

This is **CRITICAL**. If you forget this line, it won't show up on the DriverStation as an OpMode to select from. Any line that starts with an `@` is called an Annotation. You can choose from `@Teleop()` or `@Autonomous()`. You can optionally give it a name and a group, but if you leave those off then it will use your class name as the name. This works well enough, so we'll typically leave those pieces out. Another annotation that you'll see commonly is `@Disabled` If you have that, then your code will compile but it won't be shown in the list of OpModes.[3]

```
7  public class HelloWorld extends OpMode {
```

```
17 }
```

`public` - means others can see it. Required for OpModes. We'll discuss this more in section 5.3.

`class` - means we are defining a class <name> should be the same as the filename. By convention, it should be started with a capital letter and each new word is a capital letter (Pascal case). We'll talk more about classes in chapter 5.

`extends OpMode` - This means the class is a child of OpMode . A child gets all of the behavior of its parent and then can add (or replace) functionality. We'll talk about what this means in chapter 14.

a class is defined from the opening curly brace "`{`" to the closing curly brace "`}`"

```
8      @Override
9      public void init() {
10         telemetry.addData("Hello","World");
11     }
```

`@Override` tells the compiler that we are meaning to override (replace) functionality in our parent class. We'll talk more about this in chapter 14.

`public` means this method is callable from outside the class. We'll discuss this more in section 5.3

`void` means it doesn't return anything. We'll talk about return types in subsection 5.2.1

[3] Our team often does that for test code that we don't want to distract us during a tournament but is VERY helpful to have where we can make it available quickly.

`init` is the name of a method. We'll talk more about methods in section 5.2

Inside of the parenthesis are any parameters passed in or none. (as in this case) We'll talk about parameters in subsection 5.2.2

The method is defined from the opening curly brace "`{`" to the closing curly brace "`}`"

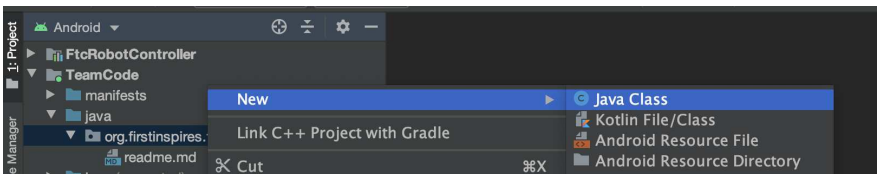
`telemetry.addData(caption, value);` - This is very cool because it sends data back to the driver station which lets us debug problems. In this case we sent back a string, but you can also send back numbers or variables. You'll notice that this ends in a semi-colon "`;`" All statements in JAVA either end with a semi-colon or have a set of curly braces attached.

```
13      @Override
14      public void loop() {
15
16      }
```

This looks much the same as our `init()` method, but there is no code in the `loop()` method, so the program won't do anything here.

1.3. Now you try

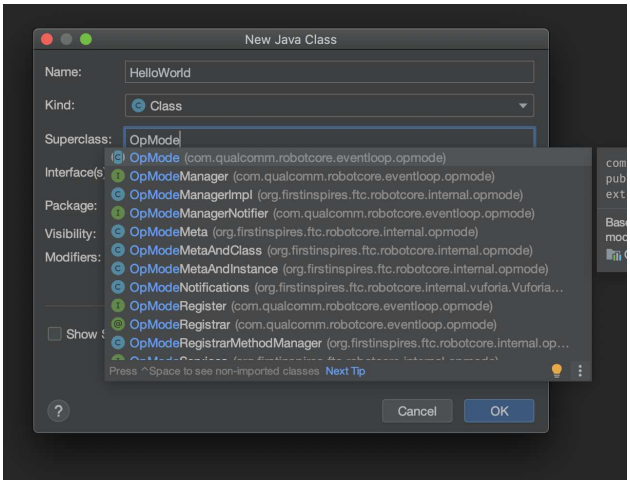
You'll learn the best here if you type in the examples (and you'll get faster at Android Studio). While this may seem like it slows you down, it helps you learn faster. To start, with change the project area to show "Android" (by using the dropdown.) If you are wondering why your Android Studio is white colored while mine is Dark, that is because I use the built-in theme "Darcula".[4]



1. Right click on org.firstinspires.ftc.teamcode

2. Select New > Java Class.

[4]To change your theme click File > Settings from the menu bar (or Android Studio > Preferences on macOS). Go to Appearance under Appearance and Behavior, and you'll see Theme.



3. Fill in the name as HelloWorld
4. Fill in the Superclass as OpMode. (We'll explain what this means in chapter 14) As you type it in, it will show you the matches. When you select it, it will fill in as `com.qualcomm.robotcore.eventloop.opmode`
5. Press "OK"

You'll get a file that will be like this:

```
package org.firstinspires.ftc.teamcode;
import com.qualcomm.robotcore.eventloop.opmode.OpMode;

public class HelloWorld extends OpMode {
}
```

It will have a red squiggle line under the class declaration. That is because you haven't implemented the two required methods yet. You haven't done anything wrong.

Make yours look like the HelloWorld.java file above. (You can start at line 6 and you'll watch it make the import statements as you type)

As you start typing, you'll notice that Android Studio is giving suggestions. You can either click on the one you want, or when it is at the top of the list then press tab.

This is the same pattern you'll follow for all OpModes in this book. (and in your robot)

1.4. Comments

So far our programs have been only for the computer. But it turns out that you can put things in them that are only for the human readers. You can (and should) add comments to the program which the computer ignores and are for human readers only. Comments should explain things that are not obvious from the code such as why something is being done. In general, comments should explain why and not what. Please don't just put in a comment what the code is doing.

Java supports two forms of comments:

1. A single line comment. It starts with a `//` and tells the computer to ignore the rest of the line.

```
// This is a comment
```

2. The block comment style. It starts with a `/*` and continues until a `*/` is encountered. This can cross multiple lines. Below are three examples.

```
/* This is also a comment */

/* So is this */

/*
* And
* this
* as
* well */
```

In addition, there is a subset of this type of comment called a javadoc that we'll talk about in chapter 15. This starts on a line with a `/**` and then goes until it sees `*/`. This is used for automatically creating documentation from your comments.

```
/**
* This is a javadoc comment
*/
```

Here is what it looks like with comments added.

Listing 1.2: `HelloWorldCommented.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class HelloWorldCommented extends OpMode {
8      /**
9       * This is called when the driver presses INIT
10      */
11     @Override
12     public void init() {
13         // this sends to the driver station
14         telemetry.addData("Hello","World");
15     }
16
17     /**
18      * This is called repeatedly while OpMode is playing
19      */
20     @Override
```

```
21      public void loop() {
22          // intentionally left blank
23      }
24  }
```

1.5. Sending to the Robot Controller

1. Make sure your phones are setup as it describes in the FTC document and that they can see each other.
2. Connect the Robot Controller to the computer.
3. Press the green play arrow next to the name of the device on the top toolbar.
4. Wait until you hear the sound from the Robot Controller and the Driver Station.
5. Now press the right arrow on the driver station to see the list of TeleOp OpModes. (The arrow on the left shows the list of Autonomous OpModes)
6. Select HelloWorld, and then press the big INIT button.
7. You should see “Hello: World” in the area where the Telemetry data is reported.

1.6. Gotchas

If your program won’t compile (or it doesn’t do what you expect), here are a few things to check that often confuse people:

- The programming language is case sensitive. In other words, `myVar` is different than `MyVar`
- Whitespace (spaces, tabs, blank lines) is all collapsed to the equivalent of a single space. It is for the human reader only.
- Blocks of code are encapsulated with curly braces ’`{`’ and ’`}`’
- Every open parenthesis ’`(`’ must have a matching close parenthesis ’`)`’
- Each program statement needs to end with a semicolon ’`;`’. In general, this means that each line of your program will have a semicolon. Exceptions are:
  - Semicolons (like everything) are ignored in comments
  - Semicolons are not used after the end curly brace. ’`}`’

1.7. Exercises

After you have done the exercise, send it to the robot controller to make sure it works.

There are sample solutions in Appendix C. However, you should struggle with them first and only look there when you are stuck. If you end up looking there, you should make up another exercise for yourself.

1. Change the code so that instead of saying “Hello: World” it says Hello and then your name.

2. Change the OpMode so it shows up in the Autonomous section of the Driver Station instead of the Teleop setion.

2. Variables and Data Types

A variable is a named location in memory where we can store information. We name variables starting with a lower case letter and then every word after that starts with a capital letter. For example: `motorSpeed` or `gyroHeading`. In Java, we specify what type of information we are storing. *Primitive datatypes* are types that are built-in to Java.

We must declare a variable before we can use it. Declaring a variable requires that we specify the type and name. It is always followed by a `;` (semi-colon).

```
// datatype name
int teamNumber;
double motorSpeed;
boolean touchSensorPressed;
```

The above variable types are `int`, `double`, and `boolean`. (These are the three you'll use most often in FTC) We'll discuss these and the other primitive datatypes in the next section.

In Java, if you don't assign a value to a variable when you create it then it starts out being equal to 0. (or `false` for `boolean`)

To assign a value to a variable, you use the = operator like this:

```
teamNumber = 16072;
motorSpeed = 0.5;
touchSensorPressed = true;
```

You can assign a value to a variable multiple times and it will be equal to what you assigned it to most recently.

It's common to declare a variable and assign the value in one line!

For example, to assign 0.5 to a variable named motorSpeed of type double, we write:

```
double motorSpeed = 0.5;
```

2.1. Primitive Data Types

There are 8 primitive data types in Java:

1. `byte`- from the range -128 to 127
2. `char` - for holding a single unicode character
3. `short` - a smaller integer. (almost never used in FTC)
4. `int` - this is short for integer. It is for numbers with no decimal.[1]

[1] It is also limited in the range from +2,147,483,647 to -2,147,483,648

5. `long` - this is a larger integer. You can use it when you are concerned about running out of room in an int.[2]
6. `float` - this is for floating point numbers. It is smaller than a `double` so we typically convert to a `double`.
7. `double` - this is for floating point numbers. It can hold numbers with decimals.[3]
8. `boolean` - this can be either `true` or `false`

In the code below, there are examples of the three most typical primitive types for FTC.

Listing 2.1: `PrimitiveTypes.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class PrimitiveTypes extends OpMode {
8      @Override
9      public void init() {
10         int teamNumber = 16072;
11         double motorSpeed = 0.5;
12         boolean touchSensorPressed = true;
13
14         telemetry.addData("Team Number", teamNumber);
15         telemetry.addData("Motor Speed", motorSpeed);
16         telemetry.addData("Touch Sensor", touchSensorPressed);
17     }
18
19     @Override
20     public void loop() {
21
22     }
23 }
```

In the three lines below you'll see them defined. Notice how they all follow the same pattern:

```
10         int teamNumber = 16072;
11         double motorSpeed = 0.5;
12         boolean touchSensorPressed = true;
```

They are sent to the driver station using `telemetry.addData`. Again, you'll notice that they all follow the same pattern.

```
14         telemetry.addData("Team Number", teamNumber);
15         telemetry.addData("Motor Speed", motorSpeed);
16         telemetry.addData("Touch Sensor", touchSensorPressed);
```

[2] It is limited in the range from +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808
[3] while technically it is limited, it is so large you can think of it as unlimited

2.2. String

A `String` is for holding text. You might be wondering why it is capitalized when all of the other data types we have seen so far isn't. This is because `String` is really a class. We'll talk more about classes in chapter 5. You'll notice that the pattern here is similar with `datatype variableName;` or `datatype variableName = initialValue;`

In the code below, there is an example of using a `String` data type.

Listing 2.2: `UseString.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class UseString extends OpMode {
8      @Override
9      public void init() {
10         String myName = "Alan Smith";
11
12         telemetry.addData("Hello", myName);
13     }
14
15     @Override
16     public void loop() {
17
18     }
19 }
```

2.3. Scope

This may seem unimportant, but you'll see why it matters later. A variable is only usable within its scope. Its scope is from where it is declared until the end of the block it is defined within. A block is defined as any set of open and close curly braces. `{ }`

As an unusual example:

```
public void loop(){
   int x = 5;
   // x is visible here
   {
      int y = 4;
      // x and y are visible here
   }
   // only x is visible here
}
```

2.4. Exercises

1. Change the `String` to have your name instead of mine in the code in section 2.2
2. Add a variable of type `int` that is called `grade` that has your grade in it. Use telemetry to send that to the driver station.

3. Gamepad and basic math

We can access the gamepads connected to the driver station from our OpMode. They are of the Gamepad class. We'll talk more about classes in chapter 5. Since there are two of them, they are called `gamepad1` and `gamepad2`.[1] The buttons on the gamepad are all `boolean` (`true` if they are pressed, `false` if they aren't). The joysticks are `double` with values between -1.0 and 1.0 (0.0 means in the center). There is one for each `x` (side to side) and one for each `y`(up and down). For strange reasons, up is negative and down is positive. The left trigger and right trigger are also `double` with values between 0.0 and 1.0 (0.0 means not pressed, 1.0 means fully pressed). To get to these we use `variableName.memberName` Below, we show what the memberNames are for all of the parts of the gamepad. In the image below, the ones that are bolded are `double` (Sometimes we call these analog and the ones that are binary - digital)

In the code below is an example of reading the Gamepad. The reason it is in `loop()` is because we want to update the telemetry as the gamepad changes.

Listing 3.1: `GamepadOpMode.java`

```
1 package org.firstinspires.ftc.teamcode;
2
3 import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4 import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6 @TeleOp()
7 public class GamepadOpMode extends OpMode {
8     @Override
9     public void init() {
```

[1] You might be wondering where these are declared. We'll talk about that in chapter 14

```
10      }
11
12      @Override
13      public void loop() {
14          telemetry.addData("Left stick x", gamepad1.left_stick_x);
15          telemetry.addData("Left stick y", gamepad1.left_stick_y);
16          telemetry.addData("A button", gamepad1.a);
17      }
18  }
```

You have to press the "Start" and "A" on a gamepad to get the driver station to recognize gamepad1 (and "Start" and "B" for gamepad2). Once the gamepad has been recognized the gamepad icon in the upper right will be illuminated.

3.1. Basic Math

In the last section, we talked about how to read a gamepad. You probably noticed that reading the joystick gave us a number. Once something is a number, we own it. We can do any kind of math to it to get what we wanted. Below are some of the most common operators.

Math Operator	Meaning
=	assignment operator
+	addition operator
-	subtraction operator AND negative operator (So saying `-x` is the same thing as saying `(0 - x)`)
*	multiplication operator
/	division operator - be aware that if you are using integers only the whole part is kept. It is NOT rounded. For example: `5 / 2 == 2`
%	modulo operator - This gives the remainder. For example: `5 % 2 == 1`
(and)	These are parenthesis and they allow you to specify the order of operations just like in regular math. You can use these to tell the difference between `3 * (4 + 2)` or `(3 * 4) + 2`

Below is an example of how we might set the speed forward we want to go based off of the joystick. In this case we are limiting our speed from -0.5 to 0.5 and doing it backwards of the joystick.

Listing 3.2: `MathOpMode.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
```

```
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class MathOpMode extends OpMode {
8      @Override
9      public void init() {
10     }
11
12     @Override
13     public void loop() {
14         double speedForward = -gamepad1.left_stick_y / 2.0;
15         telemetry.addData("Left stick y", gamepad1.left_stick_y);
16         telemetry.addData("speed Forward", speedForward);
17     }
18 }
```

The first thing we do is create a new variable and assign to it from another variable using math.

```
14         double speedForward = -gamepad1.left_stick_y / 2.0;
```

You'll notice that then we can send that variable directly using telemetry

```
16         telemetry.addData("speed Forward", speedForward);
```

3.2. Other assignment operators

There are some shortcuts where you can combine a math operator and an assignment operator. Below are some of the most common.

Operator	Meaning	Example
`++`	increment	`x++` means the same as `x = x + 1`
`--`	decrement	`x--` means the same as `x = x - 1`
`+=`	Add and assignment	`x += 2` means the same as`x = x + 2`
`*=`	Multiply and assignment	`x *= 2` means the same as `x = x * 2`
`/=`	divide and assignment	`x /= 2` means the same as `x = x / 2`
`%=`	modulo and assignment	`x %= 2` means the same as `x = x % 2`

3.3. Exercises

1. Add telemetry to show the right stick of `gamepad1`.
2. Add telemetry to show whether the b button is pressed on `gamepad1`

3. Report to the user the difference between the left joystick y and the right joystick y on `gamepad1`.
4. Report to the user the sum of the left and right triggers on `gamepad1`.

4. Making decisions

4.1. If

So far all of our programs have executed all of the code. Control structures allow you to change which code is executed and even to execute code multiple times.

The `if` statement is the first control structure. Here is an example of a program using it:

Listing 4.1: `IfOpMode.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class IfOpMode extends OpMode {
8      @Override
9      public void init() {
10     }
11
12     @Override
13     public void loop() {
14         if(gamepad1.left_stick_y < 0){
15             telemetry.addData("Left stick", " is negative");
16         }
17
18         telemetry.addData("Left stick y", gamepad1.left_stick_y);
19     }
20 }
```

Can you figure out what this is doing?

`if` clauses start with `if(conditionalExpression)`. It then has either a single statement or a block of code. A block of code starts with an open curly brace `{`, then it has 0 or more statement, and then a close curly brace `}`. I **strongly** recommend always using a block of code instead of a single statement. The code in the block is *only* executed if the conditional expression inside the parenthesis is **true**.

There are several conditional operators that we can use:

Operator	Meaning
==	is equal to
!=	is not equal to
<	is less than
>	is greater than
<=	is less than or equal to
>=	is greater than or equal to

A common mistake is trying to test for equality with the assignment operator = instead of the equality operator ==.

Not only can we use conditional operators, we can also use a `boolean` variable to make the decision. Here is an example:

Listing 4.2: `IfOpMode2.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class IfOpMode2 extends OpMode {
8      @Override
9      public void init() {
10     }
11
12     @Override
13     public void loop() {
14         if(gamepad1.a){
15             telemetry.addData("A Button", "pressed");
16         }
17     }
18 }
```

4.2. Else

An `if` statement can have an `else` clause which handles what should be done if the `if` expression is false. That sounds confusing, but here is an example:

Listing 4.3: `IfElseOpMode.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
```

```
7  public class IfElseOpMode extends OpMode {
8      @Override
9      public void init() {
10     }
11
12     @Override
13     public void loop() {
14         if(gamepad1.left_stick_y < 0){
15             telemetry.addData("Left stick", " is negative");
16         }
17         else{
18             telemetry.addData("Left stick", " is positive");
19         }
20
21         telemetry.addData("Left stick y", gamepad1.left_stick_y);
22     }
23 }
```

4.2.1. Else if

Since an else statement can have a single statement OR a block of code we can chain them together like this:

Listing 4.4: `IfElseIfOpMode.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class IfElseIfOpMode extends OpMode {
8      @Override
9      public void init() {
10     }
11
12     @Override
13     public void loop() {
14         if(gamepad1.left_stick_y < 0.5){
15             telemetry.addData("Left stick", " is negative and large");
16         }
17         else if (gamepad1.left_stick_y < 0){
18             telemetry.addData("Left stick", " is negative and small");
19         }
20         else if (gamepad1.left_stick_y < 0.5){
21             telemetry.addData("Left stick", " is positive and small");
22         }
23         else {
24             telemetry.addData("Left stick", " is positive and large");
25         }
26     }
```

```
27 }
```

4.3. Combinations

Sometimes you want to test for more than one thing. For example, you may want to test if a variable is between two numbers. While you can use multiple `if` statements, it is often more convenient and readable to use logical combinations. There are three ways that you can combine logical conditions.

Operator	Example	Meaning
`&&`	`(A < 10) && (B > 5)`	logical AND (return TRUE if condition A AND condition B are true, otherwise return FALSE.)
`||`	`(A < 10) || (B > 5)`	logical OR (return TRUE if condition A OR condition B is true, otherwise return FALSE.)
`!`	`!(A < 10)`	logical NOT (return TRUE if condition A is false, otherwise return FALSE.)

A common mistake is accidentally using the single `&` instead of `&&` or using the single `|` instead of `||` The single versions are for doing binary arithmetic operations. That is pretty rare in your Java FTC code so we won't be talking about it in this book.

One thing that might not be obvious is that you can use these to toggle a `boolean` variables. So for example:

```
boolean bVar;

bVar = !bVar;
```

When it is delcared, `bVar` will be `false`. (Since all boolean variables are initialized to false unless you say differently.) After the line `bVar =!bVar` it will be equal to true.

4.4. While

A `while` loop is much like an if statement except for after it is done it goes back to the beginning and checks the conditional again. (and there is no else). What if we had the amount the robot had turned, but we wanted its heading (between -180 and 180). We could use code like this:

```
        while(angle > 180){
            angle -= 360;
        }
        while(angle < -180){
            angle += 360;
        }
```

The reason it takes two while clauses is because one takes care of the case where we had turned more than 180 degrees in the positive direction, and the other takes care of the case where we had turned more than 180 degrees in the negative direction.[1]

You might be tempted to write code like

```
while(gamepad1.a){
    // do something
}
```

That code won't work in an OpMode because gamepad1 is only updated in between calls to `loop()`

There is also a `do...while` loop which executes once regardless and checks the condition at the end instead of the beginning. This is pretty rare in Java FTC code but is included here for completeness. A quick example:

```
do{
  // code goes here
  a++;
}while(a < 10)
```

4.5. For

There are two types of`for` loops. The traditonal type that looks like many programming languages - `for(start; conditional; update)` The start is executed once before we begin, the conditional is checked every time before we execute, the end is done at the end of EVERY time through.

```
for(int i = 0; i < 4; i++){
  // This code will happen 4 times
}
```

This is pretty rarely used in Java for FTC but is included for completeness.

The other one is called a `for-each` that we'll talk about when we talk about arrays in chapter 13 that is more commonly used in FTC.

4.6. Exercises

1. Make a "turbo button". When `gamepad1.a`is not pressed, multiply the joystick by 0.5 and when it is pressed multiply by 1 and report to the user as Forward Speed.

2. Make a "crazy mode". When `gamepad1.a` is pressed, report X as Y and Y as X. When it isn't pressed, report the joystick as normal....

[1] If we were doing this for real, we would do it in radians. But we used degrees here to make the concept simpler

5. Class Members and Methods

A `class` is a model of something. It can contain data (members) and functions (methods). Whenever you create a `class`, it becomes a data type that people can make variables of that type. You can think of a `class` like a blueprint that can be used to make any number of identical things. (called "objects") For example, the `String` data type is a class but we can have multiple objects of type `String` in our programs.

5.1. Members

So far, we have had variables in our methods but we can also have them belong to our class. To have them belong to our class, they just need to be within the class body but outside of every method body. By convention, they are at the beginning of the class but they don't have to be. If they are in our class, then every method in our class can use them and when they get changed everyone sees the new value. However, every object (copy) has its own member variables[1]

Listing 5.1: `ClassMemberOpMode.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class ClassMemberOpMode extends OpMode {
8      boolean initDone;
9
10     @Override
11     public void init() {
12         telemetry.addData("init Done", initDone);
13         initDone = true;
14     }
15
16     @Override
17     public void loop() {
18         telemetry.addData("init Done", initDone);
19     }
20 }
```

[1] unless they are declared `static` which means they are shared between all objects of the class. We'll talk about this in section 5.5.

This is a little strange because even though `initDone` gets updated in `init()`, nothing sends it to the driver station until `loop()` gets called for the first time.

You can use the `this` keyword to unambiguously say you are referring to the class member, but if there isn't a variable with the same name in your method then you can leave it off. That would look like `this.initDone`

5.2. Class Methods

We can create new methods. A method has a return type (which is any data type), a name, and can take 0 or more parameters. A parameter is a way you can pass information into a method. Each parameter has a data type and a name. Inside the method, it is just like you had a variable defined inside the method with that data type and name. (but it received its value from whomever called the class method.)

By convention we name methods starting with a lowercase letter and then having each additional word in the name start with an uppercase letter (Camel Case) After its parameters, there is the method body which goes from the opening curly bracket `{` to the close curly bracket `}`.

Listing 5.2: `ClassMethodOpMode.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class ClassMethodOpMode extends OpMode {
8
9      @Override
10     public void init() {
11     }
12
13     double squareInputWithSign(double input){
14         double output = input * input;
15         if(input < 0){
16             output = output * -1;
17         }
18         return output;
19     }
20
21     @Override
22     public void loop() {
23         double leftAmount = gamepad1.left_stick_x;
24         double fwdAmount = -gamepad1.left_stick_y;
25
26         telemetry.addData("Before X", leftAmount);
27         telemetry.addData("Before Y", fwdAmount);
28
29         leftAmount = squareInputWithSign(leftAmount);
```

```
30          fwdAmount = squareInputWithSign(fwdAmount);
31
32          telemetry.addData("After X", leftAmount);
33          telemetry.addData("After Y", fwdAmount);
34      }
35  }
```

5.2.1. Return Types

The return type is simply the data type in front of the name. You can also say that a method doesn't return anything. In that case, instead of the data type you put the keyword `void` before the name. To return the value you use the return statement. It is simply `return <value>;` You can return a variable or a constant (typed in number, string, etc.) You can see this done in the example above. As soon as the return keyword is executed the method returns to whomever called it.

5.2.2. Parameters

You probably noticed that the name had an open parenthesis `(` after it. Then each parameter is listed like a variable (except no default assignment allowed). If there is more than one parameter, they are seperated by a comma `,` Then at the end of the parameters is a close parenthesis `)`

So some examples of methods:

```
// returnDataType name(parameters)
double squareInputWithSign(double input){
   double output = input * input;
   if(input < 0){
       output = output * -1;
    }
    return output;
}
void setMotorSpeed(double speed){
    motor.set(speed);
}
double min(double x, double y){
    if(x < y){
        return x;
    }
    return y;
}
boolean isSensorPressed(){
    return touchSensor.isPressed();
}
```

5.2.3. Special Methods: Constructors

A constructor is a special method in a Java class that has the same name as the class and it has no return type. It gets called whenever the class is initialized. (created). In Java you can have multiple constructors where each one has different parameters.

An example:

```
public class Point{
   int x;
   int y;

   public Point(int x, int y){
       this.x = x;
       this.y = y;
   }
}
```

In this case, we had to use the `this` keyword because the class member is named the same as the parameter. Sometimes people will change the parameter name instead - like this:

```
public class Point{
   int x;
   int y;

   public Point(int x_in, int y_in){
       x = x_in;
       y = y_in;
   }
}
```

Or, people that are coming from other languages will sometimes start all class members with `m_` so it looks like this:

```
public class Point{
   int m_x;
   int m_y;

   public Point(int x, int y){
       m_x = x;
       m_y = y;
   }
}
```

Personally I prefer the first option, but it is a preference. All three are legal options and will do the same thing.

5.2.4. Another special method: toString

All objects in Java have a method called `toString()` This is used whenever we convert to a string (like when we send to `telemetry.addData`) The default has the name of the

class and its hash code (typically NOT useful.) So using our Point class example from above:

```
public class Point{
   int x;
   int y;

   public Point(int x, int y){
       this.x = x;
       this.y = y;
   }
   @Override
   public String toString(){
       return "Point " + x + " " + y;
   }
}
```

You might be wondering why we use `@Override` when we are not extending another class. It turns out in Java that all classes extend the base class `Object`

5.3. Controlling access- Keep your private things private

You can also modify all class methods and members with an access modifier. (that is who can access it.) By default, members and methods are all package-private. That means that only that class and other classes in the same package (directory) can see them. The options are: (from most to least restrictive)

- `private` - It can only be seen with the class. It cannot be accessed from outside the class.
- (default - none specified) - only that class and other classes in the same package (directory) can see them
- `protected` - It can only be seen with the class, its children, and other classes in the same package (We'll talk about children in chapter 14)
- `public` - It can be seen from everywhere. (You have seen this on `init()` and `loop()` in your OpModes)

In general, you want to be as restrictive as makes sense. If you are modifying the access, it goes before the data type. (for variables) and before the return data type (for methods)

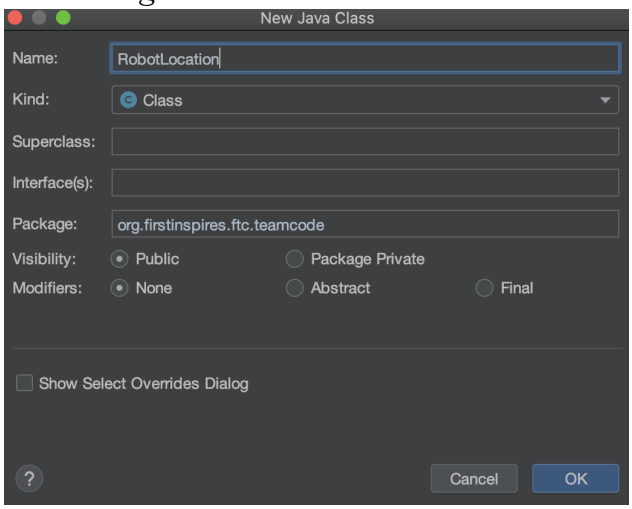
5.4. Creating your own classes

Hopefully you have been following along, so you are a pro at making your own OpMode classes by now. We start the same (remember section 1.3)

1. Right click on org.firstinspires.ftc.teamcode

2. Select New > Java Class

But in this cass we are going to name it `RobotLocation` and it will have no Superclass so the dialog should look like this:



Listing 5.3: `RobotLocation.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  public class RobotLocation{
4      double angle;
5
6      public RobotLocation(double angle){
7          this.angle = angle;
8      }
9
10     public double getHeading(){
11         double angle = this.angle;
12         while(angle > 180){
13             angle -= 360;
14         }
15         while(angle < -180){
16             angle += 360;
17         }
18         return angle;
19     }
20
21     @Override
22     public String toString(){
23         return "RobotLocation: angle (" + angle + ")";
24     }
25
26     public void turn(double angleChange){
27         angle += angleChange;
28     }
29     public void setAngle(double angle){
```

```
30          this.angle = angle;
31      }
32  }
```

Let's talk about what makes up this file.

```
4       double angle;
```

Here is an example of the class member we talked about in section 5.1 Since it doesn't have an access modifier, it is default which means it is only available to this class and other classes in the same package.

```
6       public RobotLocation(double angle){
7           this.angle = angle;
8       }
```

This is an example of a constructor liked we talked about in subsection 5.2.3. You can tell a constructor because it has no return type and it has the same name as the class. Constructors typically have the `public` access modifier so a class can be created using it from anywhere. You'll notice that it assigns the class member to a value. It uses the `this` keyword so that we can have the parameter named the same thing.

```
10      public double getHeading(){
11          double angle = this.angle;
12          while(angle > 180){
13              angle -= 360;
14          }
15          while(angle < -180){
16              angle += 360;
17          }
18          return angle;
19      }
```

This is a public class method that returns the heading (so it needs to be within -180 and 180). This would be a great place for a comment describing the method. We just left comments out of most source in the book since the text of the book comments them.

```
21      @Override
22      public String toString(){
23          return "RobotLocation: angle (" + angle + ")";
24      }
```

This is the special method `toString()` that we talked about in subsection 5.2.4. We are adding strings and numbers together here which may seem strange. The `String` class redefines (overloads) the `+` operator to mean concatenate (join) two strings together. (Yes, it also overloads `+=` to work as you would expect. No, in Java you can't overload operators in your own classes.) If it comes across something that isn't a string, it calls its `toString()` method which works (mostly) as you would expect for primitive types.

```
26      public void turn(double angleChange){
27          angle += angleChange;
28      }
```

This is a `public` class method where we can specify how much the robot is turning. You'll notice that since the parameter is not the same as the class member we are using that we don't have to use the `this` keyword for the class member. You'll also notice that we use the add and assign operator `+=` as a shortcut.

```
29      public void setAngle(double angle){
30          this.angle = angle;
31      }
```

Here is another public class method where we can set the angle.

You might have noticed that there is no way to get the angle out. (We can only get out the heading). We could absolutely add this method if we needed it.

Sometimes you'll see programmers make the class members public so they are easier to get and set. The problem with that is that it makes it hard for you to change the internals later without affecting other parts of your code. For example, right now you are keeping the angle in degrees. If you wanted to change it to be radians internally, you can do that as long as you change the methods that set it and get the heading to take and return degrees and do the conversion.

Listing 5.4: `UseRobotLocationOpMode.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp
7  public class UseRobotLocationOpMode extends OpMode {
8      RobotLocation robotLocation = new RobotLocation(0);
9
10     @Override
11     public void init() {
12         robotLocation.setAngle(0);
13     }
14
15     @Override
16     public void loop() {
17         if(gamepad1.a){
18             robotLocation.turn(0.1);
19         }
20         else if(gamepad1.b){
21             robotLocation.turn(-0.1);
22         }
23         telemetry.addData("Location", robotLocation);
24         telemetry.addData("Heading", robotLocation.getHeading());
25     }
26 }
```

This is the OpMode that uses our new class. The first 7 lines are the same so we'll start after that.

```
8       RobotLocation robotLocation = new RobotLocation(0);
```

This is a data member in our OpMode. You'll notice that it uses the `new` keyword. We use this whenever we are creating an instance of a class (or object). The `new` keyword tells the compiler to reserve room for it and call the constructor that matches the parameters your gave it. (type only, the names are ignored) Also by convention variables start with a lower case letter while the class starts with an upper case letter. They don't have to be named the same.

```
10      @Override
11      public void init() {
12          robotLocation.setAngle(0);
13      }
```

Inside our `init()` method, we call the `setAngle()` method of the `robotLocation` object. The reason we call `setAngle()` here is in case we select the opMode, init it, run it and then stop and press init again. If we don't set it in `init()` then it will keep its value from the last time it was modified.

As a best practice for FTC, your `init()` method should set things back to their expected default state.

```
15      @Override
16      public void loop() {
17          if(gamepad1.a){
18              robotLocation.turn(0.1);
19          }
20          else if(gamepad1.b){
21              robotLocation.turn(-0.1);
22          }
```

Obviously this doesn't turn the robot (because we don't have any motors hooked up), so perhaps `turn()` was an unfortunate naming choice. Run this and you'll get a feel for how fast `loop()` is called. Also, we don't allow the user to turn positively and negatively at the same time (since that makes no sense). Since it looks at `gamepad1.a` first, if they are both pressed then it will turn positively.

5.5. static

The `static` keyword means that it belongs to the type instead of the object. This can be used for methods (but then they can't access any non-static members or methods) or for class members. It is often used when you want to let someone call a method and they don't need to have an object of that type first.

5.6. Exercises

1. Add a `double getAngle()` method to `RobotLocation` and then display it in your opMode.

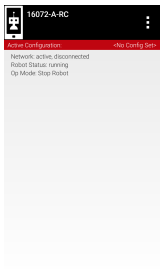
2. This exercise has two parts.
   a) Add a member of type `double` called `x` to your `RobotLocation` and add `double getX()`, `void changeX(double change)`, and `setX(double x)` methods.
   b) Change the OpMode to have `robotLocation.changeX(-0.1)` called when `gamepad1.dpad_l` is pressed and `robotLocation.changeX(0.1)` called when `gamepad1.dpad_right` is pressed
3. After you have done exercise 2, also add in support for `y`. Use `gamepad1.dpad_up` for `robotLocation.changeY(0.1)` and `gamepad1.dpad_down` for `robotLocation.changeY(-0.`

6. Our first hardware

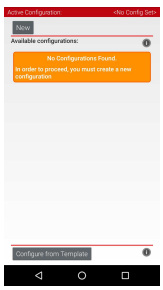
Until this point, we have been in pure software that hasn't used any of our hardware. That is fine, but our robot will be pretty boring without any sensors, motors, or servos. This (and following chapters) assume you have a programming board setup like in Appendix A

6.1. Configuration file

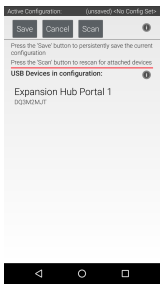
This will feel like a lot of steps the first time, but soon it'll become very natural to run through them.



1. From either the Driver Station or the Robot Controller - select the three dots in the upper right



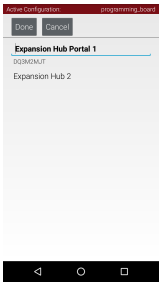
2. Select New in the upper left



3. After you press new, it should find your expansion hub. If it doesn't, please make sure your USB cable is connected between the phone and the expansion hub. (The letters and digits of your expansion hub will be unique to your hub.)

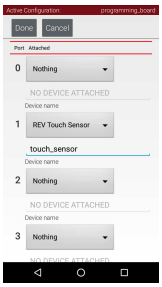
4. Press on "Expansion Hub Portal 1"

5. While you can rename it from “Expansion Hub Portal 1”, I don’t see any reason to. You will see each expansion hub that is plugged in. If you only have 1, it should say “Expansion Hub 2”. Press on it.



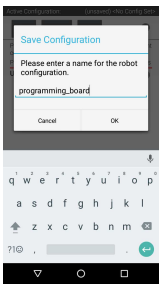
6. This will give you a listing of all of the areas where you can have communication from your REV expansion hub. Press on “Digital Devices”

7. On Port 1, Change to “REV Touch Sensor”
8. Change its name to be “touch_sensor”



9. Press Done in the upper left (going up to Expansion Hub 2)
10. Press Done again (going up to Expansion Hub Portal 1)
11. Press Done again (going up to top level)
12. Press Save

13. Change name to “programming_board”



14. Press OK
15. Press Activate under “programming_board” The upper right should now say “programming_board”
16. Press the left pointing arrow on the bottom. This will restart the robot
17. On the Driver Station, you should see “programming_board” under the image of a robot.

6.2. Mechanisms

Until this point we have had everything in one package. At this point, we are going to split things into two packages. One will hold our mechanisms (For this book, we have one mechanism called the ProgrammingBoard. On our real robot we would likely have multiple mechanisms.) The other will hold our opModes.

So there are now two programs:

This one is in the mechanisms package. To create a package, right click in the same place that we have to make a new class, but select new package and type in “mechanisms”.

That will make the package. Then right click on the package and select new class. This one should be “ProgrammingBoard1” and it should have no superclass.

Listing 6.1: `ProgrammingBoard1.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import com.qualcomm.robotcore.hardware.DigitalChannel;
4  import com.qualcomm.robotcore.hardware.HardwareMap;
5
6  public class ProgrammingBoard1 {
7      private DigitalChannel touchSensor;
8
9      public void init(HardwareMap hwMap) {
10         touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
11         touchSensor.setMode(DigitalChannel.Mode.INPUT);
12     }
13
14     public boolean getTouchSensorState() {
15         return touchSensor.getState();
16     }
17 }
```

Line 1 should be put in for you by Android Studio.

Lines 3 & 4 will be put in as you type items.

Line 6 should start out that way as you create the class

```
7      private DigitalChannel touchSensor;
```

This line says that we have a class member of type `DigitalChannel` with a name of `touchSensor`. `DigialChannel` comes from the FTC SDK. We'll talka bout how to navigate the SDK to find out what is there in chapter 16. This needs to be a class member since it is set in `init()` and used in other methods. We set it to `private` to make sure only our class can interact directly with it. This is a good practice for all hardware. Normally you would want to name it with what the sensor does (like `armInPositionTouchSensor`, but since this is part of a programming board it doesn't have more of a purpose than being a Touch Sensor.

```
9     public void init(HardwareMap hwMap) {
```

We have an `init()` method. We could have called it anything, but since we'll call it from our `init()` in our OpMode it seemed reasonable. While it might be tempting to make this the constructor, that limits what we can and can't do, so it is easier to follow the same structure. You'll notice that this takes one parameter of type `HardwareMap` and it is called `hwMap`. We could have called it `hardwareMap` but I am lazy so I took a shortcut. `HardwareMap` also comes from the FTC SDK and it is how our programs get information from the configuration file on the robot.

```
10        touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
```

This assigns to the variable touchSensor the hardware that is in the configuration file of type `DigitalChannel.class` and with a name of `touch_sensor`. This name has to match **EXACTLY** what is in the configuration file. It may seem strange to you that you don't have to use `new` here. That is because the `get()` method of `HardwareMap` does it for you.

```
11        touchSensor.setMode(DigitalChannel.Mode.INPUT);
```

It turns out that you can set each `DigitalChannel` as either`INPUT` or `OUTPUT`. Since we are reading from the touch sensor, we need to set it as an `INPUT`

```
14    public boolean getTouchSensorState() {
15        return touchSensor.getState();
16    }
```

We create a class method so that those outside of our class can read the state of the touchSensor. This is better than making `touchSensor` public because nobody can accidentally mess it up.

6.3. OpMode

This one is in the opmodes package

Listing 6.2: `TouchSensorOpMode.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
```

```
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard1;
7
8  @TeleOp()
9  public class TouchSensorOpMode extends OpMode {
10     ProgrammingBoard1 board = new ProgrammingBoard1();
11     @Override
12     public void init() {
13         board.init(hardwareMap);
14     }
15
16     @Override
17     public void loop() {
18         telemetry.addData("Touch sensor", board.getTouchSensorState());
19     }
20 }
```

The first few lines of this should look amazingly familiar by now.

```
10     ProgrammingBoard1 board = new ProgrammingBoard1();
```

Here we create a class member of type `ProgrammingBoard1` named board and we set it equal to a `new` instance of `ProgrammingBoard1` It has to be a class member so all of our methods can access it.

```
12     public void init() {
13         board.init(hardwareMap);
14     }
```

Our init is very clean. It only calls the init of our `board` object. The variable `hardwareMap` is part of the OpMode and it is how we see how the robot is configured.

```
17     public void loop() {
18         telemetry.addData("Touch sensor", board.getTouchSensorState());
19     }
```

For the loop all we do is send to the telemetry the state of the touch sensor.

6.4. Making changes

One of the huge advantages of splitting things out is that we can isolate hardware "weirdness". For example, you were probably surprised that pushing in the touch sensor returns false and it not pushed in was true. So let's change that.

First, we'll change our ProgrammingBoard class

Listing 6.3: `ProgrammingBoard2.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import com.qualcomm.robotcore.hardware.DigitalChannel;
4  import com.qualcomm.robotcore.hardware.HardwareMap;
5
```

```
6  public class ProgrammingBoard2 {
7      private DigitalChannel touchSensor;
8
9      public void init(HardwareMap hwMap) {
10         touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
11         touchSensor.setMode(DigitalChannel.Mode.INPUT);
12     }
13
14     public boolean isTouchSensorPressed() {
15         return !touchSensor.getState();
16     }
17 }
```

Since we changed the name of the method, we have to change it in the OpMode as well. (PROTIP: If we use a right click, and Refactor->Rename in Android Studio then it will magically change it both in its declaration and everywhere it is called.

Listing 6.4: `TouchSensorOpMode.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard1;
7
8  @TeleOp()
9  public class TouchSensorOpMode extends OpMode {
10     ProgrammingBoard1 board = new ProgrammingBoard1();
11     @Override
12     public void init() {
13         board.init(hardwareMap);
14     }
15
16     @Override
17     public void loop() {
18         telemetry.addData("Touch sensor", board.getTouchSensorState());
19     }
20 }
```

6.5. Exercises

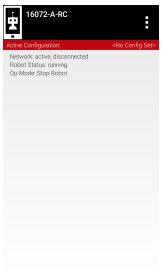
1. Add a method `isTouchSensorReleased()` to the `ProgrammingBoard2` class and use it in your opMode
2. Have your opMode send “Pressed” and “Not Pressed” for the “Touch sensor” instead of `true` or `false`. Hint, you’ll need to use your `if/else` and two `telemetry.addData` statements.

7. Motors

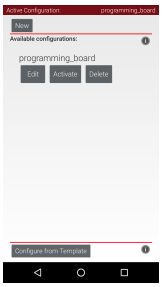
It is great that we have a sensor, but it is time to make things move!!

7.1. Editing Configuration File

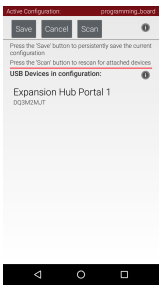
1. From either the Driver Station or the Robot Controller - select the three dots in the upper right



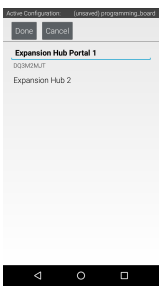
2. Press edit under the “programming_board” config that we made earlier

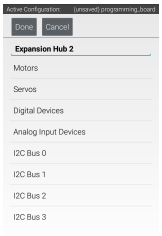


3. Press on “Expansion Hub Portal 1”

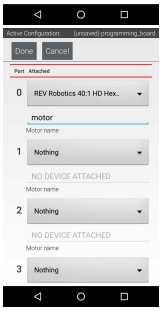


4. While you can rename it from “Expansion Hub Portal 1”, I don’t see any reason to. You will see each expansion hub that is plugged in. If you only have 1, it should say “Expansion Hub 2”. Press on it.





5. This will give you a listing of all of the areas where you can have communication from your REV expansion hub. Press on “Motors”



6. On Port 0, Change to “Rev Robotics 40:1 HD Hex Motor”
7. Change its name to be “motor”
8. Press Done in the upper left (going up to Expansion Hub 2)
9. Press Done again (going up to Expansion Hub Portal 1)
10. Press Done again (going up to top level)
11. Press Save
12. Press OK
13. Press Activate under “programming_board” The upper right should now say “programming_board”
14. Press the left pointing arrow on the bottom. This will restart the robot
15. On the Driver Station, you should see “programming_board” under the image of a robot.

7.2. Mechanisms

Listing 7.1: `ProgrammingBoard3.java`

```
1 package org.firstinspires.ftc.teamcode.mechanisms;
2
3 import com.qualcomm.robotcore.hardware.DcMotor;
4 import com.qualcomm.robotcore.hardware.DigitalChannel;
5 import com.qualcomm.robotcore.hardware.HardwareMap;
6
7 public class ProgrammingBoard3 {
8     private DigitalChannel touchSensor;
9     private DcMotor motor;
```

```
10
11      public void init(HardwareMap hwMap) {
12          touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
13          touchSensor.setMode(DigitalChannel.Mode.INPUT);
14          motor = hwMap.get(DcMotor.class, "motor");
15          motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
16      }
17      public boolean isTouchSensorPressed() {
18          return !touchSensor.getState();
19      }
20
21      public void setMotorSpeed(double speed){
22          motor.setPower(speed);
23      }
24  }
```

Most of this should look the same as our last file, so we'll just talk about the changes

```
9       private DcMotor motor;
```

Here we are adding a variable of type `DcMotor` with name `motor`. Normally you would want to name the motor with what it does, but since this is part of a programming board - we'll just call in `motor`. `DcMotor` comes from the FTC SDK.

```
14          motor = hwMap.get(DcMotor.class, "motor");
```

This assigns to the variable motor the hardware that is in the configuration file of type `DcMotor.class` and with a name of `motor`. This name has to match **EXACTLY** what is in the configuration file.

```
15          motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
```

This sets how we want to use the motor. The choices are:

RunMode	**Meaning**
RUN_TO_POSITION	The motor is to attempt to rotate in whatever direction is necessary to cause the encoder reading to advance or retreat from its current setting to the setting which has been provided through the `setTargetPosition()` method.
RUN_USING_ENCODER	The motor is to do its best to run at targeted velocity.
RUN_WITHOUT_ENCODER	The motor is simply to run at whatever velocity is achieved by apply a particular power level to the motor.
STOP_AND_RESET_ENCODER	The motor is to set the current encoder position to zero.

We set it here to `DcMotor.RunMode.RUN_USING_ENCODER` which means that it uses the encoder on the motor so that we are setting a speed and it figures out how to modify power to get to that speed (if possible). We like this mode because if you set two motors to the same speed then they have a better chance at being at the same speed than in

any other mode. (We have met teams that don't even plug in the encoders and they are having weird problems with the robot not driving straight.)

While `RUN_TO_POSITION` can be very handy for single motors, we recommend AGAINST using it in a drive train because the different speeds for the different wheels trying to get to a position can cause wacky side effects.

```
21      public void setMotorSpeed(double speed){
22          motor.setPower(speed);
23      }
```

This is a class method so that code outside our class can set the speed of the motor. This is better than exposing the `motor` as public because people can't accidentally change configuration.

7.3. OpMode

This one is in the opmodes package

Listing 7.2: `MotorOpMode.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard4;
7
8  @TeleOp()
9  public class MotorOpMode extends OpMode {
10     ProgrammingBoard4 board = new ProgrammingBoard4();
11     @Override
12     public void init() {
13         board.init(hardwareMap);
14     }
15
16     @Override
17     public void loop() {
18         board.setMotorSpeed(0.5);
19     }
20 }
```

This has very little that is new, so we'll only talk about that.

```
17      public void loop() {
18          board.setMotorSpeed(0.5);
19      }
```

Here we don't do anything conditional. We just set the motor to a speed of 0.5 (half way forwards) Technically we could have had a `start()` method that did this but since

we have to have a `loop()` in our OpMode anyway, we went for the simple. Yes, it will tell the motor to go to the same speed over and over. It doesn't matter.

7.4. Motor as Sensor

The motor also has a rotation sensor built into it. We are using it when we say RUN_USING_ENC but we can also read it and use it in our code. It'll need a chance to the Programming-Board file

Listing 7.3: `ProgrammingBoard4.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import com.qualcomm.robotcore.hardware.DcMotor;
4  import com.qualcomm.robotcore.hardware.DigitalChannel;
5  import com.qualcomm.robotcore.hardware.HardwareMap;
6
7  public class ProgrammingBoard4 {
8      private DigitalChannel touchSensor;
9      private DcMotor motor;
10     private double ticksPerRotation;
11
12     public void init(HardwareMap hwMap) {
13         touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
14         touchSensor.setMode(DigitalChannel.Mode.INPUT);
15         motor = hwMap.get(DcMotor.class, "motor");
16         motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
17         ticksPerRotation = motor.getMotorType().getTicksPerRev();
18     }
19     public boolean isTouchSensorPressed() {
20         return !touchSensor.getState();
21     }
22
23     public void setMotorSpeed(double speed){
24         motor.setPower(speed);
25     }
26     public double getMotorRotations(){
27         return motor.getCurrentPosition() / ticksPerRotation;
28     }
29 }
```

Most of this is the same so we'll just talk about the differences

```
10     private double ticksPerRotation;
```

This is a member variable where we will store the number of encoder ticks per rotation. We do this to make things easier for the opModes.

```
17         ticksPerRotation = motor.getMotorType().getTicksPerRev();
```

If we set the exact motor we have in the configuration, then we can do this to get the number of ticks per rev (revolution). I prefer to call them rotation since our students come from FLL teams where they are more used to that terminology. If you have additional gear changes after the motor, you'll have to put this in manually.

```
26      public double getMotorRotations(){
27          return motor.getCurrentPosition() / ticksPerRotation;
28      }
```

This is a class method where we return the number of motor rotations. To get the number of rotations from the number of encoder ticks, we simply divide the number of ticks by the number of ticks in a rotation. One nice thing about Java is that if there is math between an `int` and a `double`, the result will be a `double`. (However, be warned that dividing an int by an int always gives an int result even if it doesn't divide equally.)

Listing 7.4: `MotorOpMode2.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard4;
7
8  @TeleOp()
9  public class MotorOpMode2 extends OpMode {
10     ProgrammingBoard4 board = new ProgrammingBoard4();
11     @Override
12     public void init() {
13         board.init(hardwareMap);
14     }
15
16     @Override
17     public void loop() {
18         board.setMotorSpeed(0.5);
19         telemetry.addData("Motor rotations", board.getMotorRotations());
20     }
21 }
```

This only has one line added from before

```
19          telemetry.addData("Motor rotations", board.getMotorRotations());
```

Here we are simply sending to telemetry what we are seeing from the motor rotations.

If your encoder counts are not going up when you are sending a positive speed to your motor, you probably have the power wires flipped going to the motor.

7.5. Motors and Sensors together

We don't need to make any change to our configuration file or our ProgrammingBoard file since they already have a motor and a sensor.

Listing 7.5: `MotorSensorOpMode.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard4;
7
8  @TeleOp()
9  public class MotorSensorOpMode extends OpMode {
10     ProgrammingBoard4 board = new ProgrammingBoard4();
11     @Override
12     public void init() {
13         board.init(hardwareMap);
14     }
15
16     @Override
17     public void loop() {
18         if(board.isTouchSensorPressed()) {
19             board.setMotorSpeed(0.5);
20         }
21         else{
22             board.setMotorSpeed(0.0);
23         }
24         telemetry.addData("Motor rotations", board.getMotorRotations());
25     }
26 }
```

Remember that setting the motor speed to 0 makes it stop. You can set for each motor what you would like it to do when set to zero by calling `setZeroBehavior()` with either `DcMotor.ZeroPowerBhavior.BRAKE` or `DcMotor.ZeroPowerBhavior.FLOAT`

So in this case, when the touch sensor is pressed we move the motor "forward" at half speed. When it isn't, we stop it.

You may end up in a circumstance where you want "forward" to be the opposite direction of clockwise. (Like on the left hand side of your drive train). To do this,you simply call the motor's method `setDirection()` with `DcMotorSimple.Direction.REVERSE` and if you want to change it back you call it with `DcMotorSimple.Direction.FORWARD`. The motor remembers these settings.

So you might make your ProgrammingBoard class `init()` method look like this:

```
...
motor = hwMap.get(DcMotor.class, "motor");
motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
motor.setZeroPowerBehavior(DcMotor.ZeroPowerBehavior.BRAKE);
motor.setDirection(DcMotorSimple.Direction.REVERSE);
```

```
ticksPerRotation = motor.getMotorType().getTicksPerRev();
...
```

7.6. Motors and Gamepads

And of course, we can use our Gamepad just like a sensor. (we are sensing what the human is doing.)

Listing 7.6: `MotorGamepadOpMode.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard4;
7
8  @TeleOp()
9  public class MotorGamepadOpMode extends OpMode {
10     ProgrammingBoard4 board = new ProgrammingBoard4();
11     @Override
12     public void init() {
13         board.init(hardwareMap);
14     }
15
16     @Override
17     public void loop() {
18         if(gamepad1.a) {
19             board.setMotorSpeed(0.5);
20         }
21         else{
22             board.setMotorSpeed(0.0);
23         }
24         telemetry.addData("Motor rotations", board.getMotorRotations());
25     }
26 }
```

This is exactly the same as before except for using `gamepad1.a` instead of the touch sensor.

or we can make it finer controlled by using an analog input from the gamepad

Listing 7.7: `MotorGamepadOpMode2.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard4;
7
```

```
8  @TeleOp()
9  public class MotorGamepadOpMode2 extends OpMode {
10     ProgrammingBoard4 board = new ProgrammingBoard4();
11     @Override
12     public void init() {
13         board.init(hardwareMap);
14     }
15
16     @Override
17     public void loop() {
18         double motorSpeed = gamepad1.left_stick_y;
19
20         board.setMotorSpeed(motorSpeed);
21
22         telemetry.addData("Motor speed", motorSpeed);
23         telemetry.addData("Motor rotations", board.getMotorRotations());
24     }
25 }
```

Yes, we could have used `gamepad1.left_stick_y` twice instead of making a `motorSpeed` variable. But I prefer to do it this way in case I want to do any math on the `motorSpeed` before using it.

7.7. Exercises

1. Add a method to the ProgrammingBoard that allows you to change the `ZeroPowerBehavio` of the motor, and then add to your OpMode where pressing `gamepad1.a` sets it to `BRAKE` and `gamepad1.b` sets it to `FLOAT`.

2. Make the joystick less sensitive in the middle without losing range by bringing in the `squareInputWithSign()` method from section 5.2into your opMode and using it.

8. Servos

8.1. Configuration File

Follow steps 1-5 of section 7.1, but select Servos

6. On Port 0, Change to “Servor”
7. Change its name to be “servo”

Continue with steps 8 and on of section 7.1

8.2. Mechanisms

Listing 8.1: `ProgrammingBoard5.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import com.qualcomm.robotcore.hardware.DcMotor;
4  import com.qualcomm.robotcore.hardware.DigitalChannel;
5  import com.qualcomm.robotcore.hardware.HardwareMap;
6  import com.qualcomm.robotcore.hardware.Servo;
7
8  public class ProgrammingBoard5 {
9      private DigitalChannel touchSensor;
10     private DcMotor motor;
11     private double ticksPerRotation;
12     private Servo servo;
13
14     public void init(HardwareMap hwMap) {
15         touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
16         touchSensor.setMode(DigitalChannel.Mode.INPUT);
17         motor = hwMap.get(DcMotor.class, "motor");
18         motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
19         ticksPerRotation = motor.getMotorType().getTicksPerRev();
20         servo = hwMap.get(Servo.class, "servo");
21     }
22     public boolean isTouchSensorPressed() {
23         return !touchSensor.getState();
24     }
```

```
25
26      public void setMotorSpeed(double speed){
27          motor.setPower(speed);
28      }
29      public double getMotorRotations(){
30          return motor.getCurrentPosition() / ticksPerRotation;
31      }
32      public void setServoPosition(double position){
33          servo.setPosition(position);
34      }
35  }
```

This is very similar to the ones before. We'll just talk about the new parts.

```
12      private Servo servo;
```

Here we create a class member of type `Servo` named `servo`. The `Servo` class comes from the FTC SDK. Again, we would use a more descriptive name on our robot.

```
20          servo = hwMap.get(Servo.class, "servo");
```

This assigns to the variable `servo` the hardware that is in the configuration file of type `Servo.class` and with a name of `servo`. This name has to match **EXACTLY** what is in the configuration file.

```
32      public void setServoPosition(double position){
33          servo.setPosition(position);
34      }
```

This allows code outside of our class to set the servo position. Typically we might expose a method for each position we want it to go to - for example `setClawOpen()` and `setClawClose()`

`servo.setPosition()` takes a double which is a fraction between 0.0 and 1.0 saying where in that range to move. We can programmatically change what that means with two methods:

1. `servo.setDirection(Servo.Direction.REVERSE);` - this flips your range. (and yes you can also call it with `Servo.Direction.FORWARD` to flip it back)
2. `servo.scaleRange(double min, double max);`- this sets the logical min and max. Then `servo.setPosition()` is a fraction between that range. [1] It is relative to the entire range, so you can set it back with `servo.scaleRange(0.0, 1.0)`.

As an example, you might have this in the `init()` method

```
...
servo = hwMap.get(Servo.class, "servo");
servo.setDirection(Servo.Direction.REVERSE);
servo.scaleRange(0.5, 1.0);  // only go from midpoint to far right point
...
```

[1] The min has to be less than the max, so you can't use this to flip the direction.

8.3. OpMode

This one is in the opmodes package

Listing 8.2: ServoGamepadOpMode.java

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard5;
7
8  @TeleOp()
9  public class ServoGamepadOpMode extends OpMode {
10     ProgrammingBoard5 board = new ProgrammingBoard5();
11     @Override
12     public void init() {
13         board.init(hardwareMap);
14     }
15
16     @Override
17     public void loop() {
18         if(gamepad1.a) {
19             board.setServoPosition(1.0);
20         }
21         else if(gamepad1.b){
22             board.setServoPosition(0.0);
23         }
24         else{
25             board.setServoPosition(0.5);
26         }
27     }
28 }
```

The only new thing here is:

```
17     public void loop() {
18         if(gamepad1.a) {
19             board.setServoPosition(1.0);
20         }
21         else if(gamepad1.b){
22             board.setServoPosition(0.0);
23         }
24         else{
25             board.setServoPosition(0.5);
26         }
27     }
```

You'll see that we are using chained if and else so that we only try to set the servo position to one location. Otherwise we will confuse the servo and you'll likely see some jitter on it. (although the last one will likely win since there is more time in between calls

to `loop()` than within `loop()`

8.4. Exercises

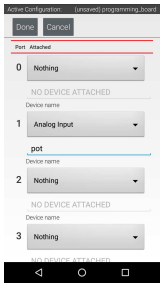
1. Change the ProgrammingBoard class so that the servo is backwards and only goes from the midpoint to far left.
2. Change the opMode so that how far you push in `gamepad1.left_trigger` determines the position of the servo.

9. Analog Sensors

We'll be using a potentiometer here, but the same concepts work for all analog sensors. It is very common to abbreviate potentiometer as "pot" because potentiometer is hard to spell.

9.1. Configuration File

Follow steps 1-5 of section 7.1, but select Analog Input Devices



6. On Port 0, Change to "Analog Input"
7. Change its name to be "pot"

Continue with steps 8 and on of section 7.1

9.2. Mechanisms

Listing 9.1: `ProgrammingBoard6.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import com.qualcomm.robotcore.hardware.AnalogInput;
4  import com.qualcomm.robotcore.hardware.DcMotor;
5  import com.qualcomm.robotcore.hardware.DigitalChannel;
6  import com.qualcomm.robotcore.hardware.HardwareMap;
7  import com.qualcomm.robotcore.hardware.Servo;
8  import com.qualcomm.robotcore.util.Range;
9
10 public class ProgrammingBoard6 {
11     private DigitalChannel touchSensor;
12     private DcMotor motor;
13     private double ticksPerRotation;
14     private Servo servo;
15     private AnalogInput pot;
16
17     public void init(HardwareMap hwMap) {
18         touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
19         touchSensor.setMode(DigitalChannel.Mode.INPUT);
```

```
20          motor = hwMap.get(DcMotor.class, "motor");
21          motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
22          ticksPerRotation = motor.getMotorType().getTicksPerRev();
23          servo = hwMap.get(Servo.class, "servo");
24          pot = hwMap.get(AnalogInput.class, "pot");
25      }
26      public boolean isTouchSensorPressed() {
27          return !touchSensor.getState();
28      }
29
30      public void setMotorSpeed(double speed){
31          motor.setPower(speed);
32      }
33      public double getMotorRotations(){
34          return motor.getCurrentPosition() / ticksPerRotation;
35      }
36      public void setServoPosition(double position){
37          servo.setPosition(position);
38      }
39      public double getPotAngle(){
40         return Range.scale(pot.getVoltage(), 0, pot.getMaxVoltage(), 0, 270);
41      }
42  }
```

Most of this is the same, so we'll just explain the new bits.

```
15      private AnalogInput pot;
```

We are declaring a class member of type `AnalogInput` with name `pot`. The `AnalogInput` class comes from the FTC SDK.

```
24          pot = hwMap.get(AnalogInput.class, "pot");
```

This assigns to the variable `pot` the hardware that is in the configuration file of type `AnalogInput.class` and with a name of `pot`. This name has to match **EXACTLY** what is in the configuration file.

```
39      public double getPotAngle(){
40         return Range.scale(pot.getVoltage(), 0, pot.getMaxVoltage(), 0, 270);
41      }
```

Thisi s a class method that returns the angle to potentiometer is currently at. It turns out that the `AnalogInput` class gives us a voltage. We could just expose that with a `getPotVoltage()` method, but then our other code has to know about voltage when it makes more sense to think in terms of the angle it is pointing at. We use a cool trick here to translate from voltage to angle.

There is a utility class in the FTC SDK called `Range` that has a method called `scale()`. It will translate a number from one range to another one. So for example if you call

```
double output = Range.scale(25, 0, 100, 0.0, 1.0);
```

then it would figure out that the input (25) was 1/4 of the way between 0 and 100. It would then figure out what 1/4 between 0 and 1.0 is and would set `output` to 0.25.

In this case we know that the lowest possible voltage that could be detected is 0, the higest we can get by calling `pot.getMaxVoltage()`. We know our potentiometer can be between 0 and 270 degrees. So we use Range.scale to convert for us.

You might have noticed that you made a method call on a class instead of an object (a variable of type class). That is because it is a `static` method. We'll talk about that in section 5.5.

9.3. OpMode

This one is in the opmodes package

Listing 9.2: `PotOpMode.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard5;
7  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard6;
8
9  @TeleOp()
10 public class PotOpMode extends OpMode {
11     ProgrammingBoard6 board = new ProgrammingBoard6();
12     @Override
13     public void init() {
14         board.init(hardwareMap);
15     }
16
17     @Override
18     public void loop() {
19         telemetry.addData("Pot Angle", board.getPotAngle());
20     }
21 }
```

Since we are doing the conversion in our ProgrammingBoard class, this becomes trivial. We are simply reporting the angle. This can be used on our robot to know what angle something is turned to.

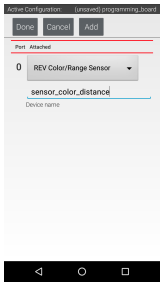
9.4. Exercises

1. Make a class method for your ProgrammingBoard that exposes the pot in the range [0.0..1.0]
2. Now make an OpMode that sets the servo to the position that the pot is returning in that range. Then you can turn the pot and it will cause the servo to "follow" it.

10. Color and Distance Sensors

10.1. Configuration File

Follow steps 1-5 of section 7.1, but select I2C Bus 1



6. On Port 0, Change to “REV Color/Range Sensor”
7. Change its name to be “sensor_color_distance”

Continue with steps 8 and on of section 7.1

10.2. Mechanisms

Listing 10.1: `ProgrammingBoard7.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import com.qualcomm.robotcore.hardware.AnalogInput;
4  import com.qualcomm.robotcore.hardware.ColorSensor;
5  import com.qualcomm.robotcore.hardware.DcMotor;
6  import com.qualcomm.robotcore.hardware.DigitalChannel;
7  import com.qualcomm.robotcore.hardware.DistanceSensor;
8  import com.qualcomm.robotcore.hardware.HardwareMap;
9  import com.qualcomm.robotcore.hardware.Servo;
10 import com.qualcomm.robotcore.util.Range;
11
12 import org.firstinspires.ftc.robotcore.external.navigation.DistanceUnit;
13
14 public class ProgrammingBoard7 {
15     private DigitalChannel touchSensor;
16     private DcMotor motor;
17     private double ticksPerRotation;
18     private Servo servo;
19     private AnalogInput pot;
20     private ColorSensor colorSensor;
21     private DistanceSensor distanceSensor;
22
23     public void init(HardwareMap hwMap) {
24         touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
```

```
25          touchSensor.setMode(DigitalChannel.Mode.INPUT);
26          motor = hwMap.get(DcMotor.class, "motor");
27          motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
28          ticksPerRotation = motor.getMotorType().getTicksPerRev();
29          servo = hwMap.get(Servo.class, "servo");
30          pot = hwMap.get(AnalogInput.class, "pot");
31
32          colorSensor = hwMap.get(ColorSensor.class, "sensor_color_distance");
33          distanceSensor = hwMap.get(DistanceSensor.class, "↩
               ↪ sensor_color_distance");
34      }
35      public boolean isTouchSensorPressed() {
36          return !touchSensor.getState();
37      }
38
39      public void setMotorSpeed(double speed){
40          motor.setPower(speed);
41      }
42      public double getMotorRotations(){
43          return motor.getCurrentPosition() / ticksPerRotation;
44      }
45      public void setServoPosition(double position){
46          servo.setPosition(position);
47      }
48      public double getPotAngle(){
49         return Range.scale(pot.getVoltage(), 0, pot.getMaxVoltage(), 0, 270);
50      }
51      public int getAmountRed(){
52          return colorSensor.red();
53      }
54      public void turnOnColorSensorLight(boolean on){
55          colorSensor.enableLed(on);
56      }
57      public double getDistance(DistanceUnit du){
58          return distanceSensor.getDistance(du);
59      }
60  }
```

Most of this is similar so we'll only talk about the new parts.

```
20      private ColorSensor colorSensor;
21      private DistanceSensor distanceSensor;
```

This is a little different. A REV ColorSensor can act as both a color sensor and a distance sensor.[1] So we make two variables - one for the `ColorSensor` class and one for the `DistanceSensor` class. Both of these classes are in the FTC SDK.

```
32          colorSensor = hwMap.get(ColorSensor.class, "sensor_color_distance");
33          distanceSensor = hwMap.get(DistanceSensor.class, "↩
               ↪ sensor_color_distance");
```

[1] Although the distance sensor is much less accurate and over a smaller range than a REV Distance sensor.

Both of these follow the pattern we have seen before. The unusual part is that they use the SAME string for the sensor. Again, it has to match EXACTLY what is in the configuration file.

```
51      public int getAmountRed(){
52          return colorSensor.red();
53      }
```

This is a class method that returns the amount of red that the color sensor sees (between 0 and 255) . The colorSensor class has several class methods that are useful.

Method	What it returns
red()	Amount of red seen (0-255)
green()	Amount of green seen (0-255)
blue()	Amount of blue seen (0-255)
argb()	An integer in the format #aarrggbb (where a is alpha, r is red, g is green, b is blue)

```
54      public void turnOnColorSensorLight(boolean on){
55          colorSensor.enableLed(on);
56      }
```

This is a class method where we can turn on the built-in LED to illuminate our target better to get a more accurate light reading. We can also use this as driver feedback.

```
57      public double getDistance(DistanceUnit du){
58          return distanceSensor.getDistance(du);
59      }
```

This uses a neat class included in the FTC SDK called `DistanceUnit`. It allows us to decide what units we want to work in and hopefully keeps us from making a NASA class mistake[2] This is a simple pass through so we'll talk more about `DistanceUnit` as we discuss the OpMode.

10.3. OpMode

This one is in the opmodes package

Listing 10.2: `DistanceColorOpMode.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.robotcore.external.navigation.DistanceUnit;
7  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard7;
```

[2] https://en.wikipedia.org/wiki/Mars_Climate_Orbiter

```
8
9  @TeleOp()
10 public class DistanceColorOpMode extends OpMode {
11     ProgrammingBoard7 board = new ProgrammingBoard7();
12     @Override
13     public void init() {
14         board.init(hardwareMap);
15     }
16
17     @Override
18     public void loop() {
19         board.turnOnColorSensorLight(gamepad1.a);
20
21         telemetry.addData("Amount red", board.getAmountRed());
22         telemetry.addData("Distance (CM)", board.getDistance(DistanceUnit.CM↩
               ↪ ));
23         telemetry.addData("Distance (IN)", board.getDistance(DistanceUnit.↩
               ↪ INCH));
24     }
25 }
```

A lot of this is similar, so let's talk about the new parts.

```
19         board.turnOnColorSensorLight(gamepad1.a);
```

You could argue (and I would agree) that this should have been written like:

```
if(gamepad1.a){
   board.turnOnColorSensorLight(true);
}else{
   board.turnOnColorSensorLight(false);
}
```

That would have been much clearer and obvious what was going on. But I wanted to show you this shortcut for two reasons:

1. So you won't be surprised when you see it in someone else's code
2. So you'll take pity on people reading your code and realize how much more readable the second example is.

```
21         telemetry.addData("Amount red", board.getAmountRed());
```

This simply prints the amount of red seen by the color sensor

```
22         telemetry.addData("Distance (CM)", board.getDistance(DistanceUnit.CM↩
               ↪ ));
23         telemetry.addData("Distance (IN)", board.getDistance(DistanceUnit.↩
               ↪ INCH));
```

This is showing the coolness of the `DistanceUnit` class. By passing in different values to `getDistance()`, we get it in the units we prefer. (you should prefer metric - but since a lot of the FTC specs are in Imperial, it is helpful to be able to do both.) The choices are:

Parameter	Unit
`DistanceUnit.MM`	millimeter
`DistanceUnit.CM`	centimeter
`DistanceUnit.INCH`	inch
`DistanceUnit.METER`	meter

If you are using this with your class, you'll have to decide what unit you are going to store things in (I typically recommend CM, but that is up to you.) Then you can convert things like this:

```
public class Square{
   double length_cm = 10;

   public double getLength(DistanceUnit du){
      return du.fromCm(length_cm);
   }
   public void setLength(double length, DistanceUnit du){
        length_cm = du.toCm(length);
   }
}
```

10.4. Exercises

1. Add a method `getAmountBlue()` to the ProgrammingBoard and report it back by changing OpMode
2. Make the motor stop when the distance sensor sees something closer than 10cm and go at half speed when farther than that.

11. Gyro (IMU)

11.1. Configuration File

Unlike everything else, you don't need to add it to the robot configuration because it is already there as "imu". You can rename it or delete it.

11.2. Mechanisms

Listing 11.1: `ProgrammingBoard8.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import com.qualcomm.hardware.bosch.BNO055IMU;
4  import com.qualcomm.robotcore.hardware.AnalogInput;
5  import com.qualcomm.robotcore.hardware.ColorSensor;
6  import com.qualcomm.robotcore.hardware.DcMotor;
7  import com.qualcomm.robotcore.hardware.DigitalChannel;
8  import com.qualcomm.robotcore.hardware.DistanceSensor;
9  import com.qualcomm.robotcore.hardware.HardwareMap;
10 import com.qualcomm.robotcore.hardware.Servo;
11 import com.qualcomm.robotcore.util.Range;
12
13 import org.firstinspires.ftc.robotcore.external.navigation.AngleUnit;
14 import org.firstinspires.ftc.robotcore.external.navigation.AxesOrder;
15 import org.firstinspires.ftc.robotcore.external.navigation.AxesReference;
16 import org.firstinspires.ftc.robotcore.external.navigation.DistanceUnit;
17 import org.firstinspires.ftc.robotcore.external.navigation.Orientation;
18
19 public class ProgrammingBoard8 {
20     private DigitalChannel touchSensor;
21     private DcMotor motor;
22     private double ticksPerRotation;
23     private Servo servo;
24     private AnalogInput pot;
25     private ColorSensor colorSensor;
26     private DistanceSensor distanceSensor;
27     private BNO055IMU imu;
28
29     public void init(HardwareMap hwMap) {
30         touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
31         touchSensor.setMode(DigitalChannel.Mode.INPUT);
32         motor = hwMap.get(DcMotor.class, "motor");
```

```
33          motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
34          ticksPerRotation = motor.getMotorType().getTicksPerRev();
35          servo = hwMap.get(Servo.class, "servo");
36          pot = hwMap.get(AnalogInput.class, "pot");
37
38          colorSensor = hwMap.get(ColorSensor.class, "sensor_color_distance");
39          distanceSensor = hwMap.get(DistanceSensor.class, "↩
                ↪ sensor_color_distance");
40          imu = hwMap.get(BNO055IMU.class, "imu");
41          BNO055IMU.Parameters params = new BNO055IMU.Parameters();
42          // change to default set of parameters go here
43          imu.initialize(params);
44      }
45
46      public boolean isTouchSensorPressed() {
47          return !touchSensor.getState();
48      }
49
50      public void setMotorSpeed(double speed){
51          motor.setPower(speed);
52      }
53      public double getMotorRotations(){
54          return motor.getCurrentPosition() / ticksPerRotation;
55      }
56      public void setServoPosition(double position){
57          servo.setPosition(position);
58      }
59      public double getPotAngle(){
60         return Range.scale(pot.getVoltage(), 0, pot.getMaxVoltage(), 0, 270);
61      }
62      public int getAmountRed(){
63          return colorSensor.red();
64      }
65      public void turnOnColorSensorLight(boolean on){
66          colorSensor.enableLed(on);
67      }
68      public double getDistance(DistanceUnit du){
69          return distanceSensor.getDistance(du);
70      }
71      public double getHeading(AngleUnit angleUnit) {
72          Orientation angles = imu.getAngularOrientation(AxesReference.↩
                ↪ INTRINSIC,
73                                                         AxesOrder.ZYX,
74                                                         angleUnit);
75          return angles.firstAngle;
76      }
77  }
```

The IMU (Inertial Measurement Unit) that is inside of every REV Expansion Hub and REV Control Hub is based off of the BNO055IMU (say that 5 times fast...) While it has a TON of capabilities, we are going to just barely tap into it here.

```
27    private BNO055IMU imu;
```

We create a class member of type `BNO055IMU` (you guessed it from the FTC SDK) with the name imu.

```
40        imu = hwMap.get(BNO055IMU.class, "imu");
```

First, we get the imu from the hardware map (just like we have done with other pieces of hardware). If you didn't change the name in your configuration (and you shouldn't), it will be "imu".

```
41        BNO055IMU.Parameters params = new BNO055IMU.Parameters();
42        // change to default set of parameters go here
43        imu.initialize(params);
```

Next, we create a variable of type `BNO055IMU.Parameters` (a class within a class.) named `params`. When we create it, it gets the default set of parameters. We can modify them, but in this case we don't.

Then we initialize the imu with the parameters.

```
71    public double getHeading(AngleUnit angleUnit) {
```

We are creating a class method so code outside of our class can get the heading of the robot (actually REV hub). Much like we had `DistanceUnit` before, there is also a class called `AngleUnit`. There are two angle units supported: `DEGREES` and `RADIANS`.[1] `AngleUnit` will make sure everything is normalized (that means it will be within -180 and 180 degrees for `DEGREES` and between -Π and Π for `RADIANS`.

```
72        Orientation angles = imu.getAngularOrientation(AxesReference.↩
              ↪ INTRINSIC,
73                                                       AxesOrder.ZYX,
74                                                       angleUnit);
```

The first thing I want to point out is that you can use white space to make the code more readable (like is done here.)

`imu.getAngularOrientation` takes three parameters:

1. `AxesReference` - can be either `INTRINSIC` (moves with object that is rotating) or `EXTRINSIC` (fixed with respect to the world). (Yes, Axes is the plural of Axis)
2. `AxesOrder` - what order you want the Axes returned in. We are saying we want them in the order `ZYX`. For reasons I don't understand in addition to `XYZ`, `XZY`, `ZXY`, `ZYX`, `YXZ`, `YZX` there is also `XYX`, `XZX`, `YXY`, `YZY`, `ZXZ`, `ZYZ`. If you understand why, please contact me and tell me.
3. `AngleUnit`- What unit we want the angles in. This can be either `DEGREES` or `RADIANS`.

```
75        return angles.firstAngle;
```

[1] No love for gradians... - https://en.wikipedia.org/wiki/Gradian

We return the `firstAngle` of the orientation (which will be the Z Axis since we asked for `ZYX`.)

This may seem really confusing, but the good news is that you only have to write it once and make sure it works. Then after that you can forget all of the complication and just call our class method `getHeading()`.

11.3. OpMode

Listing 11.2: `GyroOpMode.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.robotcore.external.navigation.AngleUnit;
7  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard8;
8
9  @TeleOp()
10 public class GyroOpMode extends OpMode {
11     ProgrammingBoard8 board = new ProgrammingBoard8();
12     @Override
13     public void init() {
14         board.init(hardwareMap);
15     }
16
17     @Override
18     public void loop() {
19         telemetry.addData("Our Heading", board.getHeading(AngleUnit.DEGREES)↩
               ↪ );
20     }
21 }
```

Really the only thing that is new here is our telemetry in line 19. Put it on the programming board and turn it around and watch the telemetry change.

11.4. Exercises

1. Change the OpMode to also show the heading in RADIANS as well as DEGREES
2. Make the motor stopped when our heading is 0, go negative when our heading is negative, and positive when our heading is positive. (HINT: `Range.Scale()` is your friend here)

12. Dealing with State

State is where you remember what you have done and do something different because of what you have done in the past.

12.1. A simple example

So far we have always done something depending on whether a button is currently pressed. What if you wanted it to do something when you first pressed it (such as toggle a light).

Listing 12.1: `ToggleOpMode.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard8;
7
8  @TeleOp()
9  public class ToggleOpMode extends OpMode {
10     ProgrammingBoard8 board = new ProgrammingBoard8();
11     boolean aAlreadyPressed;
12     boolean lightOn;
13     @Override
14     public void init() {
15         board.init(hardwareMap);
16     }
17
18     @Override
19     public void loop() {
20         if(gamepad1.a && !aAlreadyPressed){
21             lightOn = !lightOn;
22             board.turnOnColorSensorLight(lightOn);
23         }
24         aAlreadyPressed = gamepad1.a;
25     }
26 }
```

Let's break this down:

```
11     boolean aAlreadyPressed;
12     boolean lightOn;
```

Here we define two more class members. Since we don't initialize them and they are `boolean` they start out as `false`.

```
20          if(gamepad1.a && !aAlreadyPressed){
```

In this line we are saying if `gamepad1.a` is `true` (pressed) AND `aAlreadyPressed` is NOT`true` (false) then... (Remember that `!` means NOT. So it makes false turn to true and true turn to false.)

```
21              lightOn = !lightOn;
```

This is a common shorthand. What it does is invert the boolean value. It does exactly the same thing as this code:

```
if(lightOn){
   lightOn = false;
}else{
   lightOn = true;
}
```

Normally, I like to avoid shortcuts but in this case it is so common that most programmers would prefer the way it is done in the example.

```
22              board.turnOnColorSensorLight(lightOn);
```

This actually turns on (or off) the light. More than one programmer has forgotten this piece and been puzzled when changing the value of a variable called lightOn did not actually change the light.

```
24          aAlreadyPressed = gamepad1.a;
```

Here we set `aAlreadyPressed` to the value of `gamepad1.a`.

Let's think about how this code works. The first time a user presses the A button, it will come in and `gamepad1.a` will be `true` and `aAlreadyPressed` will be `false`. So it will toggle the `lightOn` class member and change the light. If the button is still held down the next time through, `gamepad1.a` will be `true` but so will `aAlreadyPressed` so it won't go into the `if` code block. Eventually our user gets bored and lets go of `gamepad1.a`. The first time through, `gamepad1.a` will be `false` and `aAlreadyPressed` will be `true`. But then `aAlreadyPressed` will be set to `false` and we'll be ready for our user to press `gamepad1.a` again.

12.2. Autonomous state - Example

When writing autonomous code, you want to write it as separate steps. This allows you to test out parts of it separately.

Listing 12.2: `AutoState1.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
```

```
4  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard8;
7
8  @Autonomous()
9  public class AutoState1 extends OpMode {
10     ProgrammingBoard8 board = new ProgrammingBoard8();
11     int state;
12
13     @Override
14     public void init() {
15         board.init(hardwareMap);
16         state = 0;
17     }
18
19     @Override
20     public void loop() {
21         telemetry.addData("State", state);
22         if (state == 0) {
23             board.setServoPosition(0.5);
24             if (board.isTouchSensorPressed()) {
25                 state = 1;
26             }
27         } else if (state == 1) {
28             board.setServoPosition(0.0);
29             if (!board.isTouchSensorPressed()) {
30                 state = 2;
31             }
32         } else if (state == 2) {
33             board.setServoPosition(1.0);
34             board.setMotorSpeed(0.5);
35             if (board.getPotAngle() > 90) {
36                 state = 3;
37             }
38         } else if (state == 3) {
39             board.setMotorSpeed(0.0);
40             state = 4;
41         } else {
42             telemetry.addData("Auto", "Finished");
43         }
44     }
45 }
```

Let's break this down:

```
11     int state;
```

Here we create our state variable to hold which state we are in. If we don't assign an initial value it is zero.

```
16         state = 0;
```

Since it should be zero, why do we assign it again in `init()`. Well, imagine that you test your auto. Press Stop, and then test it again. If we don't reset the variable here then it will be whatever it was at the end of your test.

```
21          telemetry.addData("State", state);
```

It is very helpful for debugging to send to the driver station what step in your auto program you are so you can figure out what is going on.

```
22          if (state == 0) {
23              board.setServoPosition(0.5);
24              if (board.isTouchSensorPressed()) {
25                  state = 1;
26              }
27          } else if (state == 1) {
```

You can see here an example of using `if/else` chaining. Also, you'll notice that when the touch sensor is pressed, we change the value of state. So the next time through we'll go to the next chain.

But there is another way...

12.2.1. Using the switch statement

In Java, if you are comparing for a number of options you can use a `switch` statement. Here is the same program rewritten with a `switch` statement.

Listing 12.3: `AutoState2.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
4  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard8;
7
8  @Autonomous()
9  public class AutoState2 extends OpMode {
10     ProgrammingBoard8 board = new ProgrammingBoard8();
11     int state;
12
13     @Override
14     public void init() {
15         board.init(hardwareMap);
16         state = 0;
17     }
18
19     @Override
20     public void loop() {
21         telemetry.addData("State", state);
22         switch (state) {
23             case 0:
24                 board.setServoPosition(0.5);
```

```
25                  if (board.isTouchSensorPressed()) {
26                      state = 1;
27                  }
28                  break;
29              case 1:
30                  board.setServoPosition(0.0);
31                  if (!board.isTouchSensorPressed()) {
32                      state = 2;
33                  }
34                  break;
35              case 2:
36                  board.setServoPosition(1.0);
37                  board.setMotorSpeed(0.5);
38                  if (board.getPotAngle() > 90) {
39                      state = 3;
40                  }
41                  break;
42              case 3:
43                  board.setMotorSpeed(0.0);
44                  state = 4;
45                  break;
46              default:
47                  telemetry.addData("Auto", "Finished");
48          }
49      }
50  }
```

You may think that since this is more lines that it is worse, but let's look at it anyway. (It is personal preference based on which you feel is more readable and you can do things with `if/else` chaining that you can't do with a `switch` statement)

```
22          switch (state) {
```

A `switch` statment is written as `switch(` variable `)`

```
23              case 0:
```

Each `case` starts with the `case` keyword followed by the constant followed by a colon `:`

```
28                  break;
```

All code is executed until it hits the `break` statement. At this point, it jumps to the closing brace of the switch statement.

If you forget to put a break statement in, it will execute the next case as well. There are reasons why you might want to intentionally do this, but if it is intentional make sure you put a comment explaining why you are doing it because most people will assume it was a mistake.

```
46              default:
```

You can (but don't have to) have a `default:` clause. This will be executed if none of the other cases were a match.

But a problem with these two programs is that if you have to put one in the middle, you have to make lots of changes. We can do better....

12.2.2. Switch with strings

Listing 12.4: `AutoState3.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
4  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard8;
7
8  @Autonomous()
9  public class AutoState3 extends OpMode {
10     ProgrammingBoard8 board = new ProgrammingBoard8();
11     String state = "START";
12
13     @Override
14     public void init() {
15         board.init(hardwareMap);
16         state = "START";
17     }
18
19     @Override
20     public void loop() {
21         telemetry.addData("State", state);
22         switch (state) {
23             case "START":
24                 board.setServoPosition(0.5);
25                 if (board.isTouchSensorPressed()) {
26                     state = "WAIT_FOR_SENSOR_RELEASE";
27                 }
28                 break;
29             case "WAIT_FOR_SENSOR_RELEASE":
30                 board.setServoPosition(0.0);
31                 if (!board.isTouchSensorPressed()) {
32                     state = "WAIT_FOR_POT_TURN";
33                 }
34                 break;
35             case "WAIT_FOR_POT_TURN":
36                 board.setServoPosition(1.0);
37                 board.setMotorSpeed(0.5);
38                 if (board.getPotAngle() > 90) {
39                     state = "STOP";
40                 }
41                 break;
```

```
42              case "STOP":
43                  board.setMotorSpeed(0.0);
44                  state = "DONE";
45                  break;
46              default:
47                  telemetry.addData("Auto", "Finished");
48          }
49      }
50  }
```

Really all we have done is change state from an integer to a String. Now our code is easier to read (called self-documenting) and it is easier to add in another state. (Win-win!!)

But now if we have a typo in a string the compiler won't catch it, and we'll have a problem in our code. What if we could have the readability of strings, but have the compiler catch typos. We can....

12.2.3. Enumerated types

Listing 12.5: `AutoState4.java`

```
1   package org.firstinspires.ftc.teamcode.opmodes;
2
3   import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
4   import com.qualcomm.robotcore.eventloop.opmode.OpMode;
5
6   import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard8;
7
8   @Autonomous()
9   public class AutoState4 extends OpMode {
10      enum State {
11          START,
12          WAIT_FOR_SENSOR_RELEASE,
13          WAIT_FOR_POT_TURN,
14          STOP,
15          DONE
16      }
17
18      ProgrammingBoard8 board = new ProgrammingBoard8();
19      State state = State.START;
20
21      @Override
22      public void init() {
23          board.init(hardwareMap);
24          state = State.START;
25      }
26
27      @Override
28      public void loop() {
29          telemetry.addData("State", state);
```

```
30          switch (state) {
31              case START:
32                  board.setServoPosition(0.5);
33                  if (board.isTouchSensorPressed()) {
34                      state = State.WAIT_FOR_SENSOR_RELEASE;
35                  }
36                  break;
37              case WAIT_FOR_SENSOR_RELEASE:
38                  board.setServoPosition(0.0);
39                  if (!board.isTouchSensorPressed()) {
40                      state = State.WAIT_FOR_POT_TURN;
41                  }
42                  break;
43              case WAIT_FOR_POT_TURN:
44                  board.setServoPosition(1.0);
45                  board.setMotorSpeed(0.5);
46                  if (board.getPotAngle() > 90) {
47                      state = State.STOP;
48                  }
49                  break;
50              case STOP:
51                  board.setMotorSpeed(0.0);
52                  state = State.DONE;
53                  break;
54              default:
55                  telemetry.addData("Auto", "Finished");
56          }
57      }
58  }
```

Let's talk through some of this. This actually works exactly the same as our first switch statement except now it is more readable (and we can't assign values to it that we aren't expecting)

```
10      enum State {
11          START,
12          WAIT_FOR_SENSOR_RELEASE,
13          WAIT_FOR_POT_TURN,
14          STOP,
15          DONE
16      }
```

We can add an accessor modifier to this so that the `enum` can be accessed outside the class, but we didn't in this case. By convention, we make all values of an `enum` ALL_CAPS. They have a comma in between each one. Most of the time, it is best to put each one on its own line but you don't have to.

This is declaring a new type called `State`. It is just like making a class. An enum is actually a special class that extends `java.lang.Enum`. So yes, you can put methods and class members in it. But you don't need to and typically don't. (So yes, you could put an enum in its own file. And yes, you can create a class inside of a class.)

```
19      State state = State.START;
```

Now instead of type `String` it is of type `State`. We initialize it to `State.START`. Note that we use the type followed by a dot `.` followed by the enum value. You probably noticed that Android Studio helped you type it in. Yet another huge benefit over a string.

```
29          telemetry.addData("State", state);
```

One of the really cool things about `enum` is that they implement `toString` automagically so when you print them you get human readable descriptions.

13. Arrays

An array can hold a fixed number of values of one type. Imagine that we had four motors on our drive train. Instead of code like:

```
DcMotor motor1;
DcMotor motor2;
DcMotor motor3;
DcMotor motor4;
```

we could have:

```
DcMotor[] motors = new DcMotor[4]
```

The pattern is:

variableType[] variableName = new variableType[arraySize];

We can access each motor with an index. The index ofaArray start with an index of 0. So it might look like this:

```
motors[0] = hwMap.get(DcMotor.class, "front_left");
motors[1] = hwMap.get(DcMotor.class, "front_right");
motors[2] = hwMap.get(DcMotor.class, "back_left");
motors[3] = hwMap.get(DcMotor.class, "back_right");
```

This may seem interesting, but not all that useful until you start using other things you have learned

```
void stopAllMotors(){
   for(int i = 0; i < 4; i++){
      motors[i].setPower(0.0);
   }
}
```

This is done so often that Java has a cool shortcut for it.

```
void stopAllMotors(){
   for(DcMotor motor : motors){
      motor.setPower(0.0);
   }
}
```

The format here is `for(` variableType variableName `:`arrayName `)`

So here is an example opMode:

Listing 13.1: `ArrayOpMode.java`

```
1 package org.firstinspires.ftc.teamcode;
2
```

```
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4
5  public class ArrayOpMode extends OpMode {
6      String[] words = {"Zeroth", "First", "Second", "Third", "Fourth", "Fifth↩
           ↪ ", "Infinity"};
7      int wordIndex;
8      double DELAY_SECS = 0.5;
9
10     double nextTime;
11
12     @Override
13     public void init() {
14         wordIndex = 0;
15     }
16
17     @Override
18     public void loop() {
19         if (nextTime < getRuntime()) {
20             wordIndex++;
21             if (wordIndex >= words.length) {
22                 wordIndex = words.length - 1;
23             }
24             nextTime = getRuntime() + DELAY_SECS;
25         }
26         telemetry.addLine(words[wordIndex]);
27     }
28 }
```

13.1. ArrayList

This is all great, but an array can't grow or shrink in size. For that there is ArrayList.

```
ArrayList<int> items = new ArrayList<>();
```

The angle brackets are new. That means the type is a "Generic". What that means is that you specify what type the class uses when you define your object. So this is creating an ArrayList that holds integers. (It could be any type including classes)

A few common methods:

```
items.add(4);    // this adds this element to the end of the list
items.get(index); // returns the element at the index of the list (starts at↩
    ↪  0)
items.clear(); // removes all items from list
items.size(); // returns the number of elements in the list

ArrayList<int> secondList = new ArrayList<>();
secondList.add(5);
secondList.add(6);
items.addAll(secondList);  // adds all elements in second list to first list
```

13.1.1. Making your own generic class

Making generic classes is not done much in FTC, but I'll include it here for completeness

```
public class MyClass<T>{
   private T member;
   public void set(T var) { member = var; }
   public T get() { return member; }
}
```

Everywhere that T is gets replaced when you use the class.

13.2. Exercises

1. Modify the opMode to send the chorus of a song you know at a fixed rate on telemetry. Once it gets to the end, it should send it again.

2. Modify your solution for exercise 1 to use ArrayList<String> instead of arrays.

14. Inheritance

In Java, when you create a class it always "inherits" from a class. If you don't use the `extends` keyword then it is inheriting from the `Object` class in Java. So what does this really do?

Let's start with a simple example and then we'll show how it can be useful in FTC. (We are going to put all of these in the org.firstinspires.ftc.teamcode package (directory))

Listing 14.1: `SuperClass.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  public class SuperClass {
4      public String a() {
5          return "a";
6      }
7
8      public String b() {
9          return "b";
10     }
11 }
```

Listing 14.2: `ChildClass.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  public class ChildClass extends SuperClass {
4      @Override
5      public String a() {
6          return "A";
7      }
8
9      public String c() {
10         return "c";
11     }
12 }
```

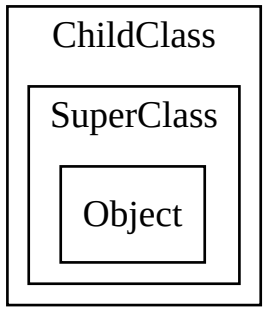
Listing 14.3: `SimpleInheritance.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4
5  public class SimpleInheritance extends OpMode {
6      SuperClass super_obj = new SuperClass();
7      ChildClass child_obj = new ChildClass();
```

```
8
9       @Override
10      public void init() {
11          telemetry.addData("Parent a", super_obj.a());
12          telemetry.addData("Parent b", super_obj.b());
13          telemetry.addData("Child a", child_obj.a());
14          telemetry.addData("Child b", child_obj.b());
15          telemetry.addData("Child c", child_obj.c());
16      }
17
18      @Override
19      public void loop() {
20
21      }
22  }
```

Can you guess what will show up on the telemetry screen? Try it. Were you right?

You can think about inheritance as your new class containing all of the super class (often called "parent") plus its new stuff. This is shown in the diagram on the right.
If you have a class method with the exact same name and parameters, then it will replace it. You should put an @Override annotation on it so that everyone knows that was intentional. (You actually don't have to but it is good practice to do it.)



14.1. Isa vs. hasa

So now there is a question. If you can get the contents of another class by either deriving from it or having it as a class member, which should you do?

This is typically called "isa" vs "hasa" (short for *is a* and *has a*) So you should derive from it if your class is of that type, but include it if you simply have it as a class member if it just just one of the things you have. Generally I like to start having it as a class member and only derive from another class if that is really clearly what I need to do.

14.2. So why in the world would you use this?

It is time for the largest word in this book - `polymorphism` - that is. When you are derived from another class you can be treated either as your class or your superclass. This will be the longest example in the book (5 files!!), but I hope it will help you take your programming to the next level.

We are going to make an OpMode that we can use to test out our wiring. (I **HIGHLY** recomend this for your robot. Once you have it, you'll find out how useful it is over

and over again to determine whether something is a software or electrical/mechanical problem.

Listing 14.4: `TestItem.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import org.firstinspires.ftc.robotcore.external.Telemetry;
4
5  abstract public class TestItem {
6      private String description;
7
8      TestItem(String description) {
9          this.description = description;
10     }
11
12     public String getDescription() {
13         return description;
14     }
15
16     abstract public void run(boolean on, Telemetry telemetry);
17 }
```

There is really only one new thing in this file but it shows up twice. It is the keyword `abstract`.

```
5  abstract public class TestItem {
```

When `abstract`is before a class it means that no objects can be made of the type of this class. (In other words it is only meant to have other classes derive from it.)

When `abstract` is before a class method it means that there is no body of this class method, but classes that derive from it that aren't abstract **MUST** implement it. (OpMode defines `init()` and `loop()` as abstract methods). Why in the world would you create a method that does nothing? Well if you require derived classes to have it, then each class can have their own implementation but you are guaranteed they have one.

Listing 14.5: `TestMotor.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import com.qualcomm.robotcore.hardware.DcMotor;
4
5  import org.firstinspires.ftc.robotcore.external.Telemetry;
6
7  public class TestMotor extends TestItem {
8      private double speed;
9      private DcMotor motor;
10
11     TestMotor(String description, double speed, DcMotor motor) {
12         super(description);
13         this.speed = speed;
```

```
14          this.motor = motor;
15      }
16
17      @Override
18      public void run(boolean on, Telemetry telemetry) {
19          if (on) {
20              motor.setPower(speed);
21          } else {
22              motor.setPower(0.0);
23          }
24          telemetry.addData("Encoder:", motor.getCurrentPosition());
25      }
26  }
```

A few notes here.

```
7   public class TestMotor extends TestItem {
```

You'll see here that this `extends` the `TestItem` class we made earlier.

```
12          super(description);
```

The `super` keyword refers to the class we derived from. Since this calls `super()` that is calling our superclass constructor. This is considered the correct way to implement a constructor in a child class.

Everything else in this file we have seen before

Listing 14.6: `TestAnalogInput.java`

```
1   package org.firstinspires.ftc.teamcode.mechanisms;
2
3   import com.qualcomm.robotcore.hardware.AnalogInput;
4   import com.qualcomm.robotcore.hardware.DcMotor;
5   import com.qualcomm.robotcore.util.Range;
6
7   import org.firstinspires.ftc.robotcore.external.Telemetry;
8
9   public class TestAnalogInput extends TestItem {
10      private AnalogInput analogInput;
11      private double min;
12      private double max;
13
14      TestAnalogInput(String description, AnalogInput analogInput, double min,↩
            ↪  double max) {
15          super(description);
16          this.analogInput = analogInput;
17          this.min = min;
18          this.max = max;
19      }
20
21      @Override
22      public void run(boolean on, Telemetry telemetry) {
23          telemetry.addData("Voltage: ", analogInput.getVoltage());
```

```
24          telemetry.addData("In Range:",
25                  Range.scale(analogInput.getVoltage(),
26                          0, analogInput.getMaxVoltage(),
27                          min, max));
28      }
29  }
```

This class should look very much like `TestMotor` to you. The one difference is we always read from the `analogInput` instead of using an `if` statement. (A rule I follow is you should only have to tell it to run a test if it causes something to change)

Listing 14.7: `ProgrammingBoard9.java`

```
1  package org.firstinspires.ftc.teamcode.mechanisms;
2
3  import com.qualcomm.hardware.bosch.BNO055IMU;
4  import com.qualcomm.robotcore.hardware.AnalogInput;
5  import com.qualcomm.robotcore.hardware.ColorSensor;
6  import com.qualcomm.robotcore.hardware.DcMotor;
7  import com.qualcomm.robotcore.hardware.DigitalChannel;
8  import com.qualcomm.robotcore.hardware.DistanceSensor;
9  import com.qualcomm.robotcore.hardware.HardwareMap;
10 import com.qualcomm.robotcore.hardware.Servo;
11 import com.qualcomm.robotcore.util.Range;
12
13 import org.firstinspires.ftc.robotcore.external.navigation.AngleUnit;
14 import org.firstinspires.ftc.robotcore.external.navigation.AxesOrder;
15 import org.firstinspires.ftc.robotcore.external.navigation.AxesReference;
16 import org.firstinspires.ftc.robotcore.external.navigation.DistanceUnit;
17 import org.firstinspires.ftc.robotcore.external.navigation.Orientation;
18
19 import java.util.ArrayList;
20
21 public class ProgrammingBoard9 {
22     private DigitalChannel touchSensor;
23     private DcMotor motor;
24     private double ticksPerRotation;
25     private Servo servo;
26     private AnalogInput pot;
27     private ColorSensor colorSensor;
28     private DistanceSensor distanceSensor;
29     private BNO055IMU imu;
30
31     public void init(HardwareMap hwMap) {
32         touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
33         touchSensor.setMode(DigitalChannel.Mode.INPUT);
34         motor = hwMap.get(DcMotor.class, "motor");
35         motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
36         ticksPerRotation = motor.getMotorType().getTicksPerRev();
37         servo = hwMap.get(Servo.class, "servo");
38         pot = hwMap.get(AnalogInput.class, "pot");
39
```

```
40          colorSensor = hwMap.get(ColorSensor.class, "sensor_color_distance");
41          distanceSensor = hwMap.get(DistanceSensor.class, "↩
                ↪ sensor_color_distance");
42          imu = hwMap.get(BNO055IMU.class, "imu");
43          BNO055IMU.Parameters params = new BNO055IMU.Parameters();
44          // change to default set of parameters go here
45          imu.initialize(params);
46      }
47
48      public boolean isTouchSensorPressed() {
49          return !touchSensor.getState();
50      }
51
52      public void setMotorSpeed(double speed) {
53          motor.setPower(speed);
54      }
55
56      public double getMotorRotations() {
57          return motor.getCurrentPosition() / ticksPerRotation;
58      }
59
60      public void setServoPosition(double position) {
61          servo.setPosition(position);
62      }
63
64      public double getPotAngle() {
65          return Range.scale(pot.getVoltage(), 0, pot.getMaxVoltage(), 0, 270)↩
                ↪ ;
66      }
67
68      public int getAmountRed() {
69          return colorSensor.red();
70      }
71
72      public void turnOnColorSensorLight(boolean on) {
73          colorSensor.enableLed(on);
74      }
75
76      public double getDistance(DistanceUnit du) {
77          return distanceSensor.getDistance(du);
78      }
79
80      public double getHeading(AngleUnit angleUnit) {
81          Orientation angles = imu.getAngularOrientation(AxesReference.↩
                ↪ INTRINSIC,
82                  AxesOrder.ZYX,
83                  angleUnit);
84          return angles.firstAngle;
85      }
86
87      public ArrayList<TestItem> getTests() {
```

```
88          ArrayList<TestItem> tests = new ArrayList<>();
89          tests.add(new TestMotor("PB Motor", 0.5, motor));
90          tests.add(new TestAnalogInput("PB Pot", pot, 0, 270));
91          return tests;
92      }
93  }
```

You'll notice that this has a new method at the end of it.

```

```

This says we will return an `ArrayList` containing elements of type `TestItem`

```
80      public double getHeading(AngleUnit angleUnit) {
```

Here we create the variable tests of type `ArrayList<TestItem>` and assign a new`ArrayList` to it. The <> is a shortcut since it is defined on the other side of our assignment.

```
81          Orientation angles = imu.getAngularOrientation(AxesReference.↩
                ↪ INTRINSIC,
82                  AxesOrder.ZYX,
```

Here we add our two new tests to it. Note that we had to have the `new` keyword and this calls their constructor. Also note that if we had three motors, we wouldn't need 3 classes - we would just have 3 copies of the line `tests.add(new TestMotor....` with a different description, speed, and motor variable.

```
83                  angleUnit);
```

and we return our list of tests.
Now for our OpMode

Listing 14.8: `TestWiring.java`

```
1  package org.firstinspires.ftc.teamcode.opmodes;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard9;
7  import org.firstinspires.ftc.teamcode.mechanisms.TestItem;
8
9  import java.util.ArrayList;
10
11 @TeleOp
12 public class TestWiring extends OpMode {
13     ProgrammingBoard9 board = new ProgrammingBoard9();
14     ArrayList<TestItem> tests;
15     boolean wasDown, wasUp;
16     int testNum;
17
18     @Override
19     public void init() {
20         board.init(hardwareMap);
```

```
21          tests = board.getTests();
22      }
23
24      @Override
25      public void loop() {
26          // move up in the list of test
27          if (gamepad1.dpad_up && !wasUp) {
28              testNum--;
29              if (testNum < 0) {
30                  testNum = tests.size() - 1;
31              }
32          }
33          wasUp = gamepad1.dpad_up;
34
35          // move down in the list of tests
36          if (gamepad1.dpad_down && !wasDown) {
37              testNum++;
38              if (testNum >= tests.size()) {
39                  testNum = 0;
40              }
41          }
42          wasDown = gamepad1.dpad_down;
43
44          //Put instructions on the telemetry
45          telemetry.addLine("Use Up and Down on D-pad to cycle through choices↩
                ↪ ");
46          telemetry.addLine("Press A to run test");
47          //put the test on the telemetry
48          TestItem currTest = tests.get(testNum);
49          telemetry.addData("Test:", currTest.getDescription());
50          //run or don't run based on a
51          currTest.run(gamepad1.a, telemetry);
52      }
53  }
```

A few things to point out here that I hope will inspire you.

```
14      ArrayList<TestItem> tests;
15      boolean wasDown, wasUp;
16      int testNum;
```

Our list of `tests` as a member variable, `wasDown` and `wasUp` (like in section 12.1) and `testNum` to keep track of which test number we are on. For `wasDown` and `wasUp`, you see a shortcut where if you have multiple variables of the same type you can define them together with a comma.

```
26          // move up in the list of test
27          if (gamepad1.dpad_up && !wasUp) {
28              testNum--;
29              if (testNum < 0) {
30                  testNum = tests.size() - 1;
31              }
```

```
32          }
33          wasUp = gamepad1.dpad_up;
34
35          // move down in the list of tests
36          if (gamepad1.dpad_down && !wasDown) {
37              testNum++;
38              if (testNum >= tests.size()) {
39                  testNum = 0;
40              }
41          }
42          wasDown = gamepad1.dpad_down;
```

This uses the gamepad1.dpad_up and gamepad1.dpad_down to let us scroll through the list of tests. (Right now there are only 2 but it should give the idea). We made the decision to "wrap" around, but you could make the decision to not wrap. It is up to you.

```
45          telemetry.addLine("Use Up and Down on D-pad to cycle through choices↩
                ↪ ");
46          telemetry.addLine("Press A to run test");
```

We haven't used `telemetry.addLine` before but it is just like `telemetry.addData` except it only has one parameter.

```
47          //put the test on the telemetry
48          TestItem currTest = tests.get(testNum);
49          telemetry.addData("Test:", currTest.getDescription());
```

This gets the test and then sends its description after "Test" with telemetry so the driver station can see what test they will be running.

```
51          currTest.run(gamepad1.a, telemetry);
```

`run()` takes a boolean for whether to run the test or not. We just pass in `gamepad1.a` directly here.

14.3. Exercises

1. Add a test for the touchSensor. you'll need a TestTouchSensor class and add it to the getTests() method in ProgrammingBoard. (No change needed to OpMode)

2. Add a test for the servo, you'll need a TestServo class - hint your constructor probably needs an "on" value and an "off" value for the servo. You'll also need to add it to the getTests()

3. Change ProgrammingBoard2 through ProgrammingBoard9 to derive from the one before it (ie, ProgrammingBoard2 extends ProgrammingBoard1) adding only what is necessary each time. Make sure all your OpModes still work!!

15. Javadoc

We talked earlier about a special kind of comment called a Javadoc. There are several huge benefits from commenting this way. The FTC SDK is commented in this way and that is what generates the documentation.

1. Android Studio will pick it up and give help to people using your classes
2. Autogenerating documentation that will amaze the judges

There are 3 places you can put a Javadoc comment.

1. Before your class
2. Before each class member
3. Before each class method

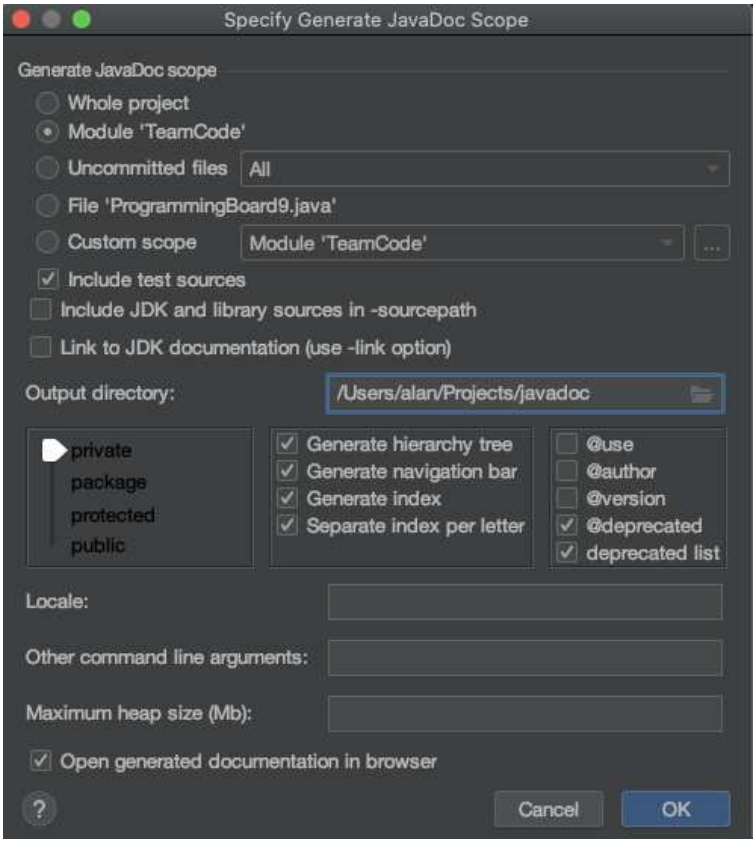
A Javadoc comment looks like this:

```
/**
* This is a javadoc comment
*/
```

If you write your class method declaration first, and then type in a `/**` above it then it will automatically put `@param` for each parameter you have and a `@return` if your method returns anything.

```
/**
  * gets our imu heading
  *
  * @param angleUnit this determines the angle unit (degrees/radians) that ↩
    ↪ it will return in
  * @return returns the current angle with the offset in the angleUnit ↩
    ↪ specified
*/
private double getHeading(AngleUnit angleUnit) {
    Orientation angles;
    angles = imu.getAngularOrientation(AxesReference.INTRINSIC,
                                       AxesOrder.ZYX,
                                       angleUnit);
     return angles.firstAngle;
}
```

After you have done this, in Android Studio go to Tools... Generate JavaDoc... and you'll see a dialog like this:



A few changes that I recommend:

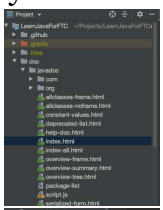
1. Do it just on Module 'TeamCode'
2. Go ahead and tell it to generate the documentation on everything
3. Make sure you put it in its own directory because it creates a lot of files

15.1. Exercises

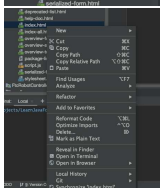
1. Add Javadoc comments to your ProgrammingBoard class
2. Add Javadoc comments to your TestMotor class (Because once you have TestWiring all working for your robot you'll want to show it to judges)

16. Finding things in FTC SDK

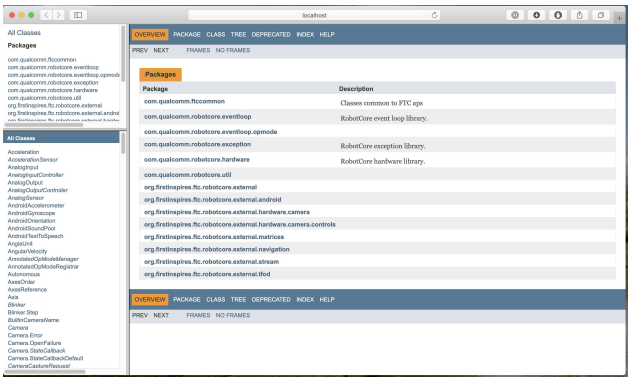
So far, I have told you about things that are in the FTC SDK. But there is lots more that we haven't looked at. So now let's teach you how to go looking for yourself.



Change the Project part to "Project" (It has been "Android" and you'll probably want to change it back after this) Look under doc>javadoc and right click on "index.html"



Select Open in Browser and pick a browser you have on your computer



You'll probably notice that this looks just like the Javadoc you created in chapter 15. Sure enough, that is what they use to create the documentation for the FTC SDK as well

For example - Look through the All Classes until you get to `Telemetry` in the lower left portion of the screen. Click on it. Then the main part of the browser will have more information out about our old friend. Wait did you see that there is a `speak()` method??

16.1. Exercise

1. Write an opMode that uses the `telemetry.speak()` method
2. Look through the documentation and find something we haven't done before and try it

17. A few other topics

This is a place for a few other topics that I thought were important to mention but didn't really fit anywhere else

17.1. Math class

The java Math class has a lot of useful methods in it. They are all static so you don't need an object of type Math. Here is an example class to handle polar coordinates

Listing 17.1: `Polar.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import org.firstinspires.ftc.robotcore.external.navigation.AngleUnit;
4
5  public class Polar {
6      double angle;
7      double magnitude;
8
9      Polar(double x, double y){
10         angle     = Math.atan2(y, x);
11         magnitude = Math.hypot(x, y);
12     }
13     double getAngle(AngleUnit angleUnit){
14         return angleUnit.fromRadians(angle);
15     }
16     double getMagnitude(){
17         return magnitude;
18     }
19 }
```

You'll notice that we have a constructor that takes in x and y and converts it to polar coordinates.

The method called getAngle uses the AngleUnit to convert. As a bonus AngleUnit guarantees results to be normalized.

Some useful methods in this class: (all trig funtions are in radians)

```
Math.abs(a)  // take the absolute value
Math.acos(a) // take the arc cosine
Math.asin(a) // take the arc sin
Math.atan(a) // take the arc tan
Math.atan2(x, y) // This returns the angle theta from conversion of ↩
    ↪ rectangular (x,y) to polar (r, theta)
```

```
Math.copySign(magnitude, sign)  // return s the first argument with the sign↩
    ↪  (positive or negative) of the second
Math.cos(a)   // take the cos
Math.hypot(x, y)  // return the sqrt(x^2 + y^2)
Math.max(a, b)  // returns the greater of a and b
Math.min(a, b)  // returns the smaller of a and b
Math.random()   // returns a double value with a positive sign greater than ↩
    ↪ or equal to 0.0 and less than 1.0
Math.signum(d)  // returns -1.0 if d < 0, 0.0 if d == 0, 1.0 if d > 0
Math.sin(a)    // take the sin
Math.sqrt(a)   // take the square root
Math.tan(a)     // take the tangent
Math.toDegrees(radians) // convert radians to degrees - I prefer using ↩
    ↪ AngleUnit
Math.toRadians(degrees) // convert degrees to radians - I prefer using ↩
    ↪ AngleUnit
```

17.2. final

`final` is a keyword that can be applied to a variable, a method or a class.

```
final int THRESHOLD = 5;
```

- `final` applied to a variable makes the variable a constant. Modifying it later will cause a compiler error. Should be initialized at this point (because you can't assign to it later.) By convention, we name these "variables" in ALL_CAPS to signify that they are constants.

```
public class SuperClass{
  public String a(){
     return "a";
  }
  final public String b(){
     return "b";
  }
}
```

- `final` applied to a method means that even if a new class `extends` this class, this method cannot be overriden

```
final class A{
  // methods and members
}
```

- `final` applied to a class means that no class can extend this one.

17.3. Exercises

1. Use the Polar class to make a new OpMode that reports the joysticks on the gamepad in polar coordinates - show the angle in degrees
2. Add the

A. Making your own Programming Board

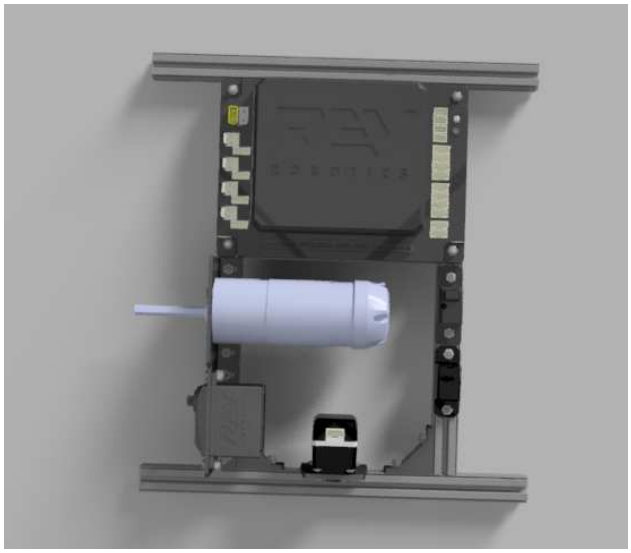
The ProgrammingBoard has a number of electrical components:

- REV Expansion Hub (http://www.revrobotics.com/rev-31-1153/)
- REV Potentiometer (http://www.revrobotics.com/rev-31-1155/)
- REV Color Sensor (http://www.revrobotics.com/rev-31-1557/)
- REV Touch Sensor (http://www.revrobotics.com/rev-31-1425/)
- REV 40:1 HD Hex Motor (http://www.revrobotics.com/rev-41-1301/)
- REV SRS Servo (http://www.revrobotics.com/rev-41-1097/)

It should be connected in the following way:

- REV 40:1 HD Hex Motor - Power and encoder to Motor 0
- REV Potentiometer - connected to Analog/Digital 0:1
- REV Color Sensor - connected to I2C 1
- REV Touch Sensor - connected to Analog/Digital 2:3
- REV SRS Servo - connected to Servo 0

Here is an example CAD from one of my students[1] of a way to assemble it using all mechanical parts from the REV FTC Kit.



[1] Thanks, Eric!!

B. LinearOpMode

B.1. What is it?

`LinearOpMode` is a class derived from `OpMode` that instead of having the five methods of an `OpMode` has only one. `runOpMode()`. Everything then occurs in that method. You are now responsible to update telemetry whenever you want it sent to the driver station, waiting for the Start button to be pressed, and checking to see if the `opModeIsActive()`

Here is our HelloWorld as a `LinearOpMode`

Listing B.1: `HelloWorldLinear.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class HelloWorldLinear extends LinearOpMode {
8
9      @Override
10     public void runOpMode(){
11         telemetry.addData("Hello","World");
12         telemetry.update();
13         waitForStart();
14         while (opModeIsActive()) {
15         }
16     }
17 }
```

So you can compare, here it is again from chapter 1

Listing B.2: `HelloWorld.java`

```
1  package org.firstinspires.ftc.teamcode;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class HelloWorld extends OpMode {
8      @Override
9      public void init() {
10         telemetry.addData("Hello","World");
11     }
12
```

```
13      @Override
14      public void loop() {
15
16      }
17  }
```

B.2. Should you use it?

My opinion is simple - ***NO!*** but since many teams do I think it is worth elaborating here why that is my opinion so you can make your own decision.

B.2.1. Benefits of LinearOpMode

The reason `LinearOpMode` exists is that it allows code to be written that is more similar to how code is often taught. Instead of using state machines like we did in chapter 12, it allows simple code like:

```
...
    board.setMotorSpeed(0.5);
    while(!board.touchSensorPressed()){
    }
    board.setMotorSpeed(0.0);
...
```

as opposed to code like:

```
...
    switch(state){
       case State.BEGIN:
           board.setMotorSpeed(0.5);
           state = State.WAIT_FOR_TOUCH;
           break;
       case State.WAIT_FOR_TOUCH:
           if(board.touchSensorPressed){
                state = State.STOP;
           }
           break;
       case State.STOP:
           board.setMotorSpeed(0.0);
           break;
...
```

The other large benefit is much of the sample code available online is written this way.

B.2.2. Drawbacks of LinearOpMode

1. `LinearOpMode` is derived from `OpMode`. If you look at the implementation of `LinearOpMode`, the `start()` method creates a thread and calls the user class `runOpMode()`. This

means you have now introduced another thread into the system. Instead of variables like gamepad being updated between calls to your OpMode, they could be updated at anytime.

2. Your code is all in one main control method instead of being broken out into logical methods.

3. You also are no longer protected from a loop taking too long so you don't respond in time to the driver station.

4. State machines are typically used in commercial embedded projects. Why not choose to learn how to do that now?

C. Sample Solutions

These are here if you get stuck, but they are not the only way to solve the exercises.

C.1. Chapter 1 Solutions

Listing C.1: `Exercise_1_1.java`

```
1  package org.firstinspires.ftc.teamcode.solutions;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp
7  public class Exercise_1_1 extends OpMode {
8      @Override
9      public void init() {
10         telemetry.addData("Hello", "Alan");
11     }
12
13     @Override
14     public void loop() {
15
16     }
17 }
```

Listing C.2: `Exercise_1_2.java`

```
1  package org.firstinspires.ftc.teamcode.solutions;
2
3  import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
4  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
5
6  @Autonomous
7  public class Exercise_1_2 extends OpMode {
8      @Override
9      public void init() {
10         telemetry.addData("Hello", "Alan");
11     }
12
13     @Override
14     public void loop() {
15
16     }
```

```
}
```

C.2. Chapter 2 Solutions

Listing C.3: Exercise_2_1.java

```
package org.firstinspires.ftc.teamcode.solutions;

import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;

@TeleOp()
public class Exercise_2_1 extends OpMode {
    @Override
    public void init() {
        String myName = "Your Name";

        telemetry.addData("Hello", myName);
    }

    @Override
    public void loop() {

    }
}
```

Listing C.4: Exercise_2_2.java

```
package org.firstinspires.ftc.teamcode.solutions;

import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;

@TeleOp()
public class Exercise_2_2 extends OpMode {
    @Override
    public void init() {
        String myName = "Your Name";
        int grade = 38;

        telemetry.addData("Hello", myName);
        telemetry.addData("Grade", grade);
    }

    @Override
    public void loop() {

    }
}
```

C.3. Chapter 3 Solutions

Listing C.5: Exercise_3_1.java

```
package org.firstinspires.ftc.teamcode.solutions;

import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;

@TeleOp()
public class Exercise_3_1 extends OpMode {
    @Override
    public void init() {
    }

    @Override
    public void loop() {
        telemetry.addData("Right stick x", gamepad1.right_stick_x);
        telemetry.addData("Right stick y", gamepad1.right_stick_y);
    }
}
```

Listing C.6: Exercise_3_2.java

```
package org.firstinspires.ftc.teamcode.solutions;

import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;

@TeleOp()
public class Exercise_3_2 extends OpMode {
    @Override
    public void init() {
    }

    @Override
    public void loop() {
        telemetry.addData("B button", gamepad1.b);
    }
}
```

Listing C.7: Exercise_3_3.java

```
package org.firstinspires.ftc.teamcode.solutions;

import com.qualcomm.robotcore.eventloop.opmode.OpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;

@TeleOp()
public class Exercise_3_3 extends OpMode {
    @Override
```

```
9     public void init() {
10    }
11
12    @Override
13    public void loop() {
14        telemetry.addData("Diff left y and right y",
15                gamepad1.left_stick_y - gamepad1.right_stick_y);
16    }
17 }
```

Listing C.8: `Exercise_3_4.java`

```
1  package org.firstinspires.ftc.teamcode.solutions;
2
3  import com.qualcomm.robotcore.eventloop.opmode.OpMode;
4  import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
5
6  @TeleOp()
7  public class Exercise_3_4 extends OpMode {
8      @Override
9      public void init() {
10     }
11
12     @Override
13     public void loop() {
14         telemetry.addData("sum triggers",
15                 gamepad1.left_trigger + gamepad1.right_trigger);
16     }
17 }
```

D. Credits

Thanks to the following people that provided feedback on earlier versions of the book to make it better.

- Karen Li (Team Techies #18175)
- Joshua Smith