Learn Java for FTC

Alan G. Smith

May 2, 2020

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

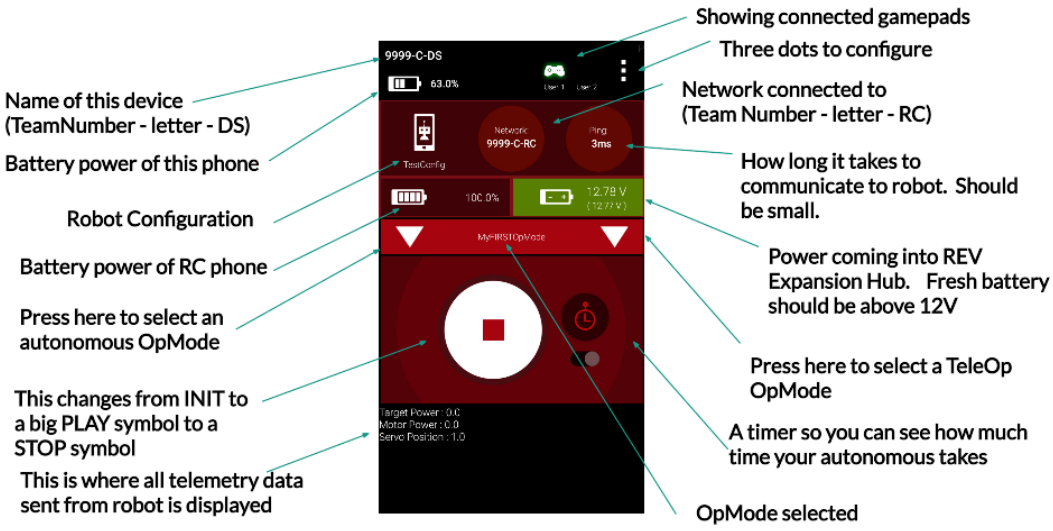
The robot controller is either an Android Phone or a REV Control Hub. It is the "brains" that are actually on your robot. We often abbrievate Robot Controller as "RC".

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.

1.1.3. 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] Go Quantum Quacks - FTC #16072

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.

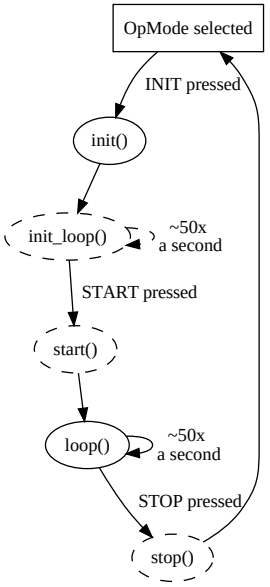
1.2.2. Parts of an OpMode[2]

In FTC, An OpMode is a program for our robot. 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. 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.)

[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.

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 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 "`}`"

[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.

```
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

`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 "`}`"

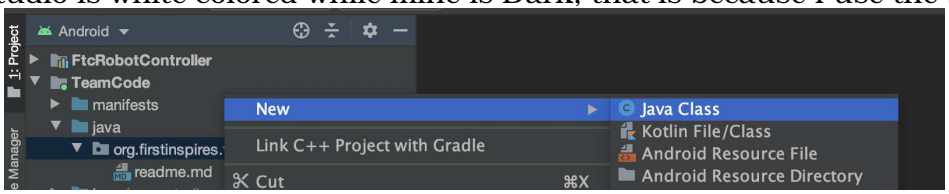
`telemetry.addData(caption, value);`- This is very cool because it sends data back to the driver station which lets us debug problems. In this case we sent back a string, but you can also send back numbers or variables. You'll notice that this ends in a semi-colon "`;`" All statements in JAVA either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

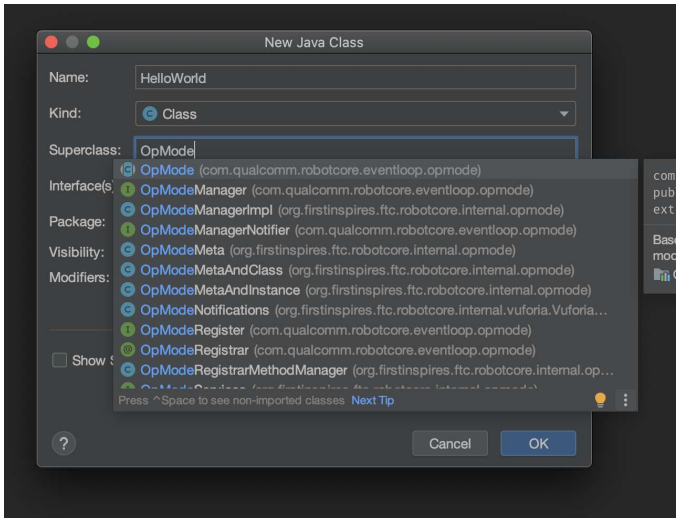
1.3. Now you try

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`
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 {
}
```

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]
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`.

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

[2] It is limited in the range from +9,223,372,036,854,775,807 to -9,223,372,036,854,775,808

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);
```

2.2. 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;
```

[3] while technically it is limited, it is so large you can think of it as unlimited

```
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() {
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);
```

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

```
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`

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

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

@TeleOp()
public class IfOpMode2 extends OpMode {
    @Override
    public void init() {
    }

    @Override
    public void loop() {
        if(gamepad1.a){
            telemetry.addData("A Button", "pressed");
        }
    }
}
```

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`

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

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

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

    @Override
    public void loop() {
        if(gamepad1.left_stick_y < 0){
            telemetry.addData("Left stick", " is negative");
        }
        else{
            telemetry.addData("Left stick", " is positive");
        }

        telemetry.addData("Left stick y", gamepad1.left_stick_y);
    }
}
```

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 create new `boolean` variables. So you can say:

```
boolean bVar;

bVar = !bVar;
```

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:

```
        double angle = this.angle;
        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]

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

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....

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 }
```

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`

[1] unless they are declared `static` which means they are shared between all objects of the class. We'll talk about this in section 18.1.

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 memory location (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)

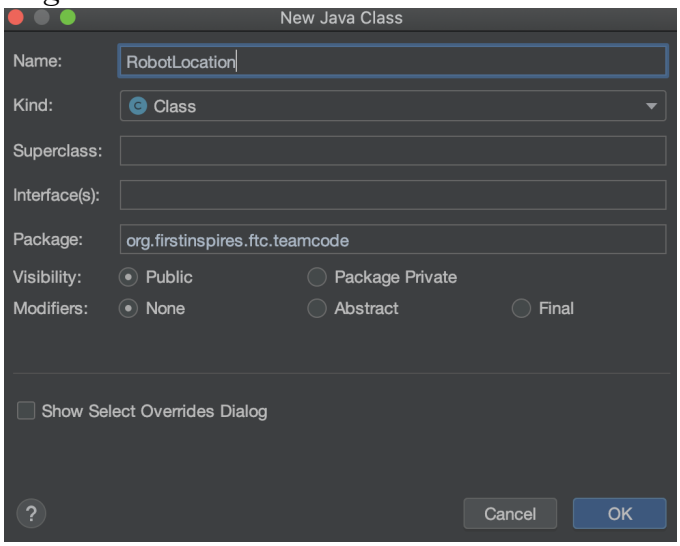
In general, you want to be as restrictive as makes sense. If you are modifying the access, it goes before the data type. (for variables) and before the return data type (for methods)

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      RobotLocation(double angle){
7          setAngle(angle);
8      }
9
10     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      void turn(double angleChange){
22          angle += angleChange;
23      }
24      void setAngle(double angle){
25          this.angle = angle;
26      }
27  }
```

And then we will need an OpMode that uses it.

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 loc = new RobotLocation(0);
9
10      @Override
11      public void init() {
12          loc.setAngle(0);
13      }
14
15      @Override
16      public void loop() {
17          if(gamepad1.a){
18              loc.turn(0.1);
19          }
20          else if(gamepad1.b){
21              loc.turn(-0.1);
22          }
23          telemetry.addData("Heading", loc.getHeading());
24      }
25  }
```

5.5. Exercises

1. Exercises need to go here

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.

6.1. Configuration file

This should talk about how to make a configuration file.

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

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 }
```

Explain the ProgrammingBoard file

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 }
```

Explain the opmodes file

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

Need to come up with exercises for here

7. Motors

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

7.1. Configuration File

This should talk about how to add a motor to the configuration file.

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 }
```

Explain the ProgrammingBoard file

7.3. OpMode

This one is in the opmodes package

Listing 7.2: `MotorOpMode.java`

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

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

import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard4;

@TeleOp()
public class MotorOpMode extends OpMode {
    ProgrammingBoard4 board = new ProgrammingBoard4();
    @Override
    public void init() {
        board.init(hardwareMap);
    }

    @Override
    public void loop() {
        board.setMotorSpeed(0.5);
    }
}
```

Explain the opmodes file

7.4. Motor as Sensor

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

Listing 7.3: `ProgrammingBoard4.java`

```java
package org.firstinspires.ftc.teamcode.mechanisms;

import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.hardware.HardwareMap;

public class ProgrammingBoard4 {
    private DigitalChannel touchSensor;
    private DcMotor motor;
    private double ticksPerRev;

    public void init(HardwareMap hwMap) {
        touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
        touchSensor.setMode(DigitalChannel.Mode.INPUT);
        motor = hwMap.get(DcMotor.class, "motor");
        motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
        ticksPerRev = motor.getMotorType().getTicksPerRev();
    }
    public boolean isTouchSensorPressed() {
        return !touchSensor.getState();
    }

    public void setMotorSpeed(double speed){
        motor.setPower(speed);
    }
    public double getMotorRotations(){
        return (double)motor.getCurrentPosition() / ticksPerRev;
    }
```

```
}
```

Explain the ProgrammingBoard file

and a chang to the OpMode

Listing 7.4: `MotorOpMode2.java`

```java
package org.firstinspires.ftc.teamcode.opmodes;

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

import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard4;

@TeleOp()
public class MotorOpMode2 extends OpMode {
    ProgrammingBoard4 board = new ProgrammingBoard4();
    @Override
    public void init() {
        board.init(hardwareMap);
    }

    @Override
    public void loop() {
        board.setMotorSpeed(0.5);
        telemetry.addData("Motor rotations", board.getMotorRotations());
    }
}
```

Explain the opmodes file

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`

```java
package org.firstinspires.ftc.teamcode.opmodes;

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

import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard4;

@TeleOp()
public class MotorSensorOpMode extends OpMode {
    ProgrammingBoard4 board = new ProgrammingBoard4();
    @Override
    public void init() {
        board.init(hardwareMap);
    }

    @Override
    public void loop() {
        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  }
```

Explain the opmodes file

7.6. Motors and Gamepads

And of course, our Gamepad is just like a sensor

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  }
```

Explain the opmodes file

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
```

```
@TeleOp()
public class MotorGamepadOpMode2 extends OpMode {
    ProgrammingBoard4 board = new ProgrammingBoard4();
    @Override
    public void init() {
        board.init(hardwareMap);
    }

    @Override
    public void loop() {
        double motorSpeed = gamepad1.left_stick_y;

        board.setMotorSpeed(motorSpeed);

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

Explain the opmodes file

7.7. Exercises

Need to come up with exercises for here

8. Servos

8.1. Configuration File

This should talk about how to add a servo to the configuration file.

8.2. Mechanisms

Listing 8.1: `ProgrammingBoard5.java`

```java
package org.firstinspires.ftc.teamcode.mechanisms;

import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.hardware.HardwareMap;
import com.qualcomm.robotcore.hardware.Servo;

public class ProgrammingBoard5 {
    private DigitalChannel touchSensor;
    private DcMotor motor;
    private double ticksPerRev;
    private Servo servo;

    public void init(HardwareMap hwMap) {
        touchSensor = hwMap.get(DigitalChannel.class, "touch_sensor");
        touchSensor.setMode(DigitalChannel.Mode.INPUT);
        motor = hwMap.get(DcMotor.class, "motor");
        motor.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
        ticksPerRev = motor.getMotorType().getTicksPerRev();
        servo = hwMap.get(Servo.class, "servo");
    }
    public boolean isTouchSensorPressed() {
        return !touchSensor.getState();
    }

    public void setMotorSpeed(double speed){
        motor.setPower(speed);
    }
    public double getMotorRotations(){
        return (double)motor.getCurrentPosition() / ticksPerRev;
    }
    public void setServoPosition(double position){
        servo.setPosition(position);
    }
}
```

Explain the ProgrammingBoard file

8.3. OpMode

This one is in the opmodes package

Listing 8.2: `MotorOpMode.java`

```java
package org.firstinspires.ftc.teamcode.opmodes;

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

import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard4;

@TeleOp()
public class MotorOpMode extends OpMode {
    ProgrammingBoard4 board = new ProgrammingBoard4();
    @Override
    public void init() {
        board.init(hardwareMap);
    }

    @Override
    public void loop() {
        board.setMotorSpeed(0.5);
    }
}
```

Explain the opmodes file

8.4. Exercises

Need to come up with exercises for here

9. Analog Sensors

9.1. Configuration File

This should talk about how to add a analog sensor to the configuration file.

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 ticksPerRev;
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         ticksPerRev = 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 (double)motor.getCurrentPosition() / ticksPerRev;
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  }
```

Explain the ProgrammingBoard file

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 }
```

Explain the opmodes file

9.4. Exercises

Need to come up with exercises for here

10. Color and Distance Sensors

10.1. Configuration File

This should talk about how to add a color sensor to the configuration file.

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 ticksPerRev;
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         ticksPerRev = 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 (double)motor.getCurrentPosition() / ticksPerRev;
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  }
```

Explain the ProgrammingBoard file

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;
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  }
```

Explain the opmodes file

10.4. Exercises

Need to come up with exercises for here

11. Gyro

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

Listing 12.1: `ToggleOpMode.java`

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

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

import org.firstinspires.ftc.robotcore.external.navigation.DistanceUnit;
import org.firstinspires.ftc.teamcode.mechanisms.ProgrammingBoard7;

@TeleOp()
public class ToggleOpMode extends OpMode {
    ProgrammingBoard7 board = new ProgrammingBoard7();
    boolean aPressed;
    boolean lightOn;
    @Override
    public void init() {
        board.init(hardwareMap);
    }

    @Override
    public void loop() {
        if(gamepad1.a && !aPressed){
            lightOn = !lightOn;
            board.turnOnColorSensorLight(lightOn);
        }
        aPressed = gamepad1.a;
    }
}
```

Explain the opmodes file

12.2. A more complicated example

This is where I will introduce switch..case and have an autonomous state machine

13. Arrays

Write this

14. Inheritance

Write this

15. Javadoc

Write this

16. Finding things in FTC SDK

Write this

17. Some useful things in the Java SDK

17.1. Math class

Write this

17.2. More of the string class

Write this

18. Other Topics (placeholder)

18.1. static

18.2. final

18.3. arrays and arrayList

A. Making your own Programming Board

Write this

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