Java for FTC Robotics

Purpose

- This document is intended to cover basic Java for FTC
- This is done by walking through files (JewelKnocker.java and Tele_Op_2017.java) from last years code
- See https://docs.oracle.com/javase/tutorial/ for more detailed information on the Java language
- See https://www.firstinspires.org/sites/default/files/uploads/resource_library/ftc/android-studio-tutorial.pdf for additional information on the FTC Java and robot system

JewelKnocker.java - Package and Imports

```
 1: package org.firstinspires.ftc.teamcode;
 2:
 3: import com.qualcomm.robotcore.hardware.DcMotor;
 4: import com.qualcomm.robotcore.hardware.Servo;
 5: import com.qualcomm.robotcore.hardware.ColorSensor;
 6: import com.qualcomm.robotcore.util.ElapsedTime;
 7:
 8: /**
 9:  * Created by jscott on 11/04/17.
10:  */
```

- JewelKnocker is a hardware class – a class created to provide a simple interface to use hardware
- The "package" statement makes this class part of the team code
- The "import" statements include FTC library classes that are used in this file.
  - Later in the file, look for where Servo, ColorSensor and ElapsedTime are used (DcMotor isn't needed)
- The last bit is just a multi-line comment. "/*" starts the comment and "*/" ends the comment

Start of Class and Class Scope Fields – key words

```
12:  public class JewelKnocker
13:  {
14:      // Enumeration for ball color
15:      public enum COLORS { RED, BLUE }
16:
17:      private Servo knockerServo = null;
18:      private ColorSensor colorSensor = null;
19:
20:      // Constants for detecting color
21:      final private static int RED_LOWER_LIMIT = 55;
22:
23:      // Position values for servo
24:      final private static double KNOCKER_DOWN_POSITION = 0.625;
25:      final private static double KNOCKER_UP_POSITION = 0.3;
26:      final private static double KNOCKER_DOWN_POSITION_8553 = 0.6;
27:      final private static double KNOCKER_UP_POSITION_8553 = 0.1;
```

- Public, private, final, class, static, enum, double, int and null are all keywords in Java
- Public means an item can be seen outside the class; private the item can only be seen / used inside the class
- Enum creates an enumeration; this creates a type that can only take the listed values (in this case BLUE and RED)
- Final means that the value of the item won't change; it is used to create constants
- Static means the value is the same across all copies of a class
- Double creates a number variable that can have a fractional part and int creates a number variable than can only contain whole numbers
- Java is an object oriented language, so items in Java are typically "instances" of a class. The class and code within in it are like the design for a house and an instance of that class is like a house built from the design

Start of Class and Class Scope Fields

```
12:  public class JewelKnocker
13:  {
14:      // Enumeration for ball color
15:      public enum COLORS { RED, BLUE }
16:
17:      private Servo knockerServo = null;
18:      private ColorSensor colorSensor = null;
19:
20:      // Constants for detecting color
21:      final private static int RED_LOWER_LIMIT = 55;
22:
23:      // Position values for servo
24:      final private static double KNOCKER_DOWN_POSITION = 0.625;
25:      final private static double KNOCKER_UP_POSITION = 0.3;
26:      final private static double KNOCKER_DOWN_POSITION_8553 = 0.6;
27:      final private static double KNOCKER_UP_POSITION_8553 = 0.1;
```

- "public class JewelKnocker" and the { on the next line is the start of the class (note that class name matches file name). Public keyword means others can see it and class indicates we are creating a class.
- The "public enum…" defines a colors type that is used to tell the user of the class what color the jewel is
- The next two "private Servo…" and "private ColorSensor…" lines define a servo motor and a color sensor object that can only be seen inside the class. Servo and ColorSensor come from robot core
- The up/down limits are constants and put here to make them easier to find / adjust. The robots require different values because they are built slightly differently.

Start of Class and Class Scope Fields (cont.)

```
12: public class JewelKnocker
13: {
14:     // Enumeration for ball color
15:     public enum COLORS { RED, BLUE }
16:
17:     private Servo knockerServo = null;
18:     private ColorSensor colorSensor = null;
19:
20:     // Constants for detecting color
21:     final private static int RED_LOWER_LIMIT = 55;
22:
23:     // Position values for servo
24:     final private static double KNOCKER_DOWN_POSITION = 0.625;
25:     final private static double KNOCKER_UP_POSITION = 0.3;
26:     final private static double KNOCKER_DOWN_POSITION_8553 = 0.6;
27:     final private static double KNOCKER_UP_POSITION_8553 = 0.1;
```

- Fields/variables usually start with a lower case letter and use mixed case for their names with the first letter of subsequent words capitalized (knocker**S**ervo)
- Names of constants are usually all CAPS with _'s between words (RED_LOWER_LIMIT)
- Java names are case sensitive (i.e. me and Me represent two different items)

Constructor

```
29:     public JewelKnocker( Servo servo, ColorSensor color )
30:     {
31:         // Store hardware references
32:         knockerServo = servo;
33:         colorSensor  = color;
34:
35:         // Turn on color sensor LED light to help make it easier to see jewels
36:         colorSensor.enableLed( true );
37:
38:         // No setup needed for servo
39:     }
```

- "public JewelKnocker…" is a constructor for the JewelKnocker class; this is why it has the same name as the class
- A constructor is a special method that runs when an instance (copy) of the class is created. It is used to initialize / setup the class.
- "Servo servo, ColorSensor color" are the input parameters to the constructor. This is the same format as other methods with a comma separated list of type and then name (i.e. Servo servo means a Servo object is passed in and the name servo is what it called in the constructor)
- "knockerServo = servo" and "colorSensor = color" copy the servo and color parameters into the private class fields.
- "colorSensor.enableLed(true)" turns on an LED that is on the color sensor. Right click on "ColorSensor" earlier in the file and choose Goto → Declaration to learn more about the library code
- Note that a ";" ends most Java statements
- This method has several line comments that start with "//". Anything from "//" to the end of the line is ignored by the compiler

Start of GetColor Method

```
41:     // Method used to determine whether color of jewel is blue or red
42:     public COLORS GetColor( )
43:     {
44:         int timesItWasRed = 0;
45:         int timesItWasBlue = 0;
46:         int tries = 0;
47:
```

- "public COLORS GetColor( )" is the beginning of a method. The body of the method starts on line 43 with the { and ends on line 80 with the matching }
- The method takes no parameters (nothing between the ( ) at the end of the method name) and returns a COLORS value (see enumeration above)
- The next 3 "int …" lines define variables used in the method and initialize them all to 0
  - These variables only exist inside GetColor
  - Variables must be defined before they are used, and can only be "seen" in the { } pair they are declared in and { } below that
  - In this case, integers were used because these are used to count. When you always want whole numbers, use an int.

While and Embedded if-else if

```
48:         while (tries < 7 )
49:         {
50:             if ( colorSensor.red( ) > colorSensor.blue() )
51:             {
52:                 timesItWasRed++;
53:                 tries++;
54:             }
55:
56:             else  if ( colorSensor.red() < colorSensor.blue() ){
57:
58:                 timesItWasBlue++;
59:                 tries++;
60:
61:             }
62:             else{
63:
64:             }
65:
66:             // Put short delay between checks
67:             Delay_ms( 20.0 );
68:
69:         }
```

- "while (tries < 7)" creates a loop that will run until tries is >= 7 (i.e. as long as the condition is true)
  - while should always be followed by a { } block – otherwise only the first line of code after the while will be run
- "if (…)… else if (…) … else" is a standard if-else if-else structure. If and else should always be followed by a { } block
- Some logical operators are greater than (>), greater than or equal (>=), less than (<), less than or equal (<=) and equal (==)
- "Delay_ms(…" uses a private method that will be described later to create a short delay

While and Embedded if-else if (cont.)

```
48:         while (tries < 7 )
49:         {
50:             if ( colorSensor.red( ) > colorSensor.blue() )
51:             {
52:                 timesItWasRed++;
53:                 tries++;
54:             }
55:
56:             else  if ( colorSensor.red() < colorSensor.blue() ){
57:
58:                 timesItWasBlue++;
59:                 tries++;
60:
61:             }
62:             else{
63:
64:             }
65:
66:             // Put short delay between checks
67:             Delay_ms( 20.0 );
68:
69:         }
```

- The <variable>++ lines increment the variable. For example, if tries was 2 and “tries++;” is executed, tries will then be 3

End of GetColor

```
70:
71:         if ( timesItWasRed >= 4 )
72:         {
73:             return COLORS.RED;
74:         }
75:         else
76:         {
77:             return COLORS.BLUE;
78:         }
79:
80:     }
```

```
COLORS jewelColor;
if ( timesItWasRed >= 4 )
{
    jewelColor = COLORS.RED;
}
else
{
    jewelColor = COLORS.BLUE;
}

return jewelColor;
```

- There is one file if/else block that determines what to return to the caller
- The method can only return COLORS.RED or COLORS.BLUE because those are the only enumerated values
- On the right is an alternate form (not in the code) that creates a COLORS variable, sets it and then returns it
- As noted earlier, line 80 closes out the method

Raise/Lower Knocker Methods

```
82:     // Method to lower the jewel knocker
83:     public void LowerKnocker( )
84:     {
85:         knockerServo.setPosition( KNOCKER_DOWN_POSITION );
86:     }
87:
88:     public void LowerKnocker8553( )
89:     {
90:         knockerServo.setPosition( KNOCKER_DOWN_POSITION_8553 );
91:     }
92:
93:     // Method to lower the jewel knocker
94:     public void RaiseKnocker( )
95:     {
96:         knockerServo.setPosition( KNOCKER_UP_POSITION );
97:     }
98:
99:     public void RaiseKnocker8553( )
100:    {
101:        knockerServo.setPosition( KNOCKER_UP_POSITION_8553 );
102:    }
103:
```

- The next 4 methods raise and lower the knocker by setting the servo position
- All methods return nothing (this is what void means) and have empty parameter lists
- There were separate methods for each team because the servos were aligned differently
- Note that the constants have to be “calibrated”

Raise/Lower Knocker Methods

```
104:     // Method used to read red value to help with debug
105:     public int RedValue( )
106:     {
107:         return colorSensor.red( );
108:     }
109:
110:     // Method used to read blue value to help with debug
111:     public int BlueValue( )
112:     {
113:         return colorSensor.blue( );
114:     }
115:
116:     // Method used to read knocker servo position for debug
117:     public double KnockerPositionGet( )
118:     {
119:         return knockerServo.getPosition( );
120:     }
121:
122:     // Method to set knocker position for debug
123:     public void KnockerPositionSet( double position )
124:     {
125:         if ( ( position >= 0.0 ) && ( position <= 1.0 ) )
126:         {
127:             knockerServo.setPosition( position );
128:         }
129:     }
```

- The next 4 methods were for debug / calibration of the jewel knocker hardware
- Note that some methods return values; these have to have a “return …” statement in them
- KnockerPositionSet is an example of a method that takes a parameter, a double named positon

Delay_ms and End of JewelKnocker

```
131:     private void Delay_ms( double delay )
132:     {
133:         ElapsedTime delayTimer = new ElapsedTime( ElapsedTime.Resolution.MILLISECONDS );
134:         delayTimer.reset();
135:         int timeWaster = 0;
136:         while ( delayTimer.time() < delay )
137:         {
138:             timeWaster++;
139:         }
140:     }
141:
142: }
```

- The final method is Delay_ms
- Delay_ms is “private” so it can only be called from within the class
- It uses ElapsedTime classes to wait a little bit
- The JewelKnocker class is closed out on line 142

Tele_Op_2017 – Package and Imports

```
33: package org.firstinspires.ftc.teamcode;
34:
Code\src\main\java\org\firstinspires\ftc\teamcode\
                                    ore.eventloop.opmode.Disabled;
36: import com.qualcomm.robotcore.eventloop.opmode.OpMode;
37: import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
38: import com.qualcomm.robotcore.hardware.ColorSensor;
39: import com.qualcomm.robotcore.hardware.DcMotor;
40: import com.qualcomm.robotcore.hardware.Servo;
41: import com.qualcomm.robotcore.util.ElapsedTime;
42: import com.qualcomm.robotcore.hardware.HardwareMap;
43:
```

- Tele_Op_2017 is an OpMode class – the top level of the robot software that we create
- Package is same as for JewelKnocker
- Imports include FTC libraries for opmodes and hardware

Tele_Op_2017 – Class Declaration

```
59:  @TeleOp(name="Tele Op 2017", group="Iterative Opmode")  // @Autonomous(...) is the other common choice
60:  // @Disabled
61:  public class Tele_Op_2017 extends OpMode
62:  {
```

- "@TeleOp(name="Tele Op 2017",…" is used to create the op mode as a tele op (driver control) on the phone. Each tele op mode MUST have a unique name (please just match the class name)
- If "@Disabled" is not commented out, the tele op mode will not show up on the phone
- "public class Tele_Op_2017 extends OpMode" creates the class and **inherits** from OpMode
  - This means that Tele_Op_2017 has all of the characteristics of OpMode like hardwareMap, gamepad1/2, etc.
  - This class must create init, init_loop, start, loop and stop methods
  - If you right click on OpMode and choose Goto → Declaration you can find more information about OpMode
- Note that this is an "Iterative Opmode", this means it is designed to loop continually without overly long loops

Tele_Op_2017 - First Fields

```
64:     /* Declare OpMode members. */
65:     private ElapsedTime runtime = new ElapsedTime();
66:
67:     // Drive hardware
68:     private DcMotor frontLeft = null;
69:     private DcMotor frontRight = null;
70:     private DcMotor rearLeft = null;
71:     private DcMotor rearRight = null;
72:
73:     // Jewel Knocker hardware
74:
75:     private Servo knockerServo = null;
76:     private ColorSensor colorSensor = null;
77:
78:
79:     HardwareMap robotMap = hardwareMap;
80:     private Drive go = null;
81:     private Pole wep = null;
82:     private Claw claw = null;
83:     private Lift lift = null;
84:
```

- Drive and JewelKnocker hardware must be created in OpMode and passed in to classes.
- Other classes use the HardwareMap and Drive and JewelKnocker could also be converted to do this
- “private Drive Go…”, “private Pole wep…”, etc. create internal fields for each of the hardware classes
- Many of the internal hardware objects are initially set to null, this means they don’t exist, yet. They will be created in “init”

Beginning of init

```
 98:     @Override
 99:     public void init() {
100:         telemetry.addData("Status", "Initializing");
101:
102:         /* eg: Initialize the hardware variables. Note that the strings used here as parameters
103:          * to 'get' must correspond to the names assigned during the robot configuration
104:          * step (using the FTC Robot Controller app on the phone).
105:          */
106:         frontLeft  = hardwareMap.dcMotor.get("front_left");
107:         frontRight  = hardwareMap.dcMotor.get("front_right");
108:         rearLeft  = hardwareMap.dcMotor.get("rear_left");
109:         rearRight  = hardwareMap.dcMotor.get("rear_right");
110:
111:
112:         knockerServo = hardwareMap.servo.get("knocker_servo");
113:         colorSensor = hardwareMap.colorSensor.get("color");
```

- Init is the first method that must be overridden (hence the "@Override"). It is run once when the user presses "Init" on the phone.
- The beginning of init uses the hardwareMap to set up the Drive and JewelKnocker hardware that must be passed to those methods
- The names passed to hardwareMap.dcMotor.get **MUST** match the names in the active configuration file on the robot

End of init

```
116:        go = new Drive(frontLeft, frontRight, rearLeft, rearRight);
117:
118:        jewelKnocker = new JewelKnocker( knockerServo, colorSensor );
119:
120:
121:        // Set up Claw
122:        claw = new Claw( hardwareMap );
123:
124:        // Set up Pole
125:        wep = new Pole( hardwareMap );
126:
127:        lift = new Lift( hardwareMap );
128:
129:        telemetry.addData("Status", "Initialized");
130:    }
```

- The rest of init creates the hardware classes and runs their constructors either with the hardware or with the hardwareMap
- The final line “telemetry.addData(…“ is how information is written out to the driver station screen

init_loop and start

```
132:    /*
133:     * Code to run REPEATEDLY after the driver hits INIT, but before they hit PLAY
134:     */
135:    @Override
136:    public void init_loop() {
137:    }
138:
139:    /*
140:     * Code to run ONCE when the driver hits PLAY
141:     */
142:    @Override
143:    public void start() {
144:        jewelKnocker.RaiseKnocker();
145:        runtime.reset();
146:    }
147:
```

- init_loop must also be overridden and runs continuously after init until the driver pushes the Play button. In this case, we chose to do nothing in init_loop.
- start must also be overridden and runs once after the Play button is pushed. To protect the jewel knocker, we put RaiseKnocker in start

Start of loop

```
148:     /*
149:      * Code to run REPEATEDLY after the driver hits PLAY but before they hit STOP
150:      */
151:     @Override
152:     public void loop() {
153:         telemetry.addData("Status", "Running: " + runtime.toString());
154:
155:         double leftClawPosition = claw.GetLeftPosition();
156:         double rightClawPosition = claw.GetRightPosition();
157:
158:         // Show joystick information as some other illustrative data
159:         telemetry.addLine("left joystick | ")
160:                 .addData("x", gamepad1.left_stick_x)
161:                 .addData("y", gamepad1.left_stick_y);
162:         telemetry.addLine("right joystick | ")
163:                 .addData("x", gamepad1.right_stick_x)
164:                 .addData("y", gamepad1.right_stick_y);
165:
166:         // Show joystick information as some other illustrative data
167:         telemetry.addLine("left joystick2 | ")
168:                 .addData("x", gamepad2.left_stick_x)
169:                 .addData("y", gamepad2.left_stick_y);
170:         telemetry.addLine("right joystick2 | ")
171:                 .addData("x", gamepad2.right_stick_x)
172:                 .addData("y", gamepad2.right_stick_y);
```

- loop is the final method that must be overridden. It runs repeatedly after Play is pressed.
- The beginning of the method mainly updates debug data on the screen
- Remember – this method runs repeatedly during Play

loop wep and lift control

```
185:        // Use gamepad Y & A raise and lower the arm
186:        wep.lift( gamepad2.right_stick_x );
187:
188:
189:        lift.Raise(gamepad2.left_stick_y);
190:
191:        // Use gamepad X & B to extend and retract the arm
192:        if (gamepad2.dpad_up)
193:        {
194:            wep.extend();
195:        }
196:        else if (gamepad2.dpad_down)
197:        {
198:            wep.retract();
199:        }
200:        else if ( gamepad2.dpad_right )
201:        {
202:            wep.extendFast();
203:        }
204:        else if ( gamepad2.dpad_left )
205:        {
206:            wep.retractFast();
207:        }
208:        else
209:        {
210:            wep.stay();
211:        }
212:
```

- This portion of loop uses gamepad inputs to control the weird extending pole (wep) and lift (big scissor lift).
- Mapping the wep extend function to the D-pad made it necessary to use the if-else if-else if-…-else structure due to how the methods work.
- Code in loop is run continually once PLAY is pressed. This means that the robot runs through this software typically in < 1 sec.
- The joysticks are continuous inputs from -1.0 to 1.0 and output 0 when not pushed. The dpad buttons just return a true or false depending on if the particular button is pressed, so the software must call “wep.stay( )” when no buttons are pressed to get the pole to stop moving.

Drive/move in loop

```
214:        // Move robot based on joystick inputs from gamepad 1 / driver 1
215:        robotForwardBack = JoystickUtilities.ShapeCubePlusInputWeighted( gamepad1.left_stick_y, ROBOT_FWD_BACK_WEIGHTING );
216:        robotLeftRight = JoystickUtilities.ShapeCubePlusInputWeighted( gamepad1.left_stick_x, ROBOT_LEFT_RIGHT_WEIGHTING );
217:        robotRotate = JoystickUtilities.ShapeCubePlusInputWeighted( gamepad1.right_stick_x, ROBOT_ROTATE_WEIGHTING );
218:        go.MoveSimple( robotLeftRight, robotForwardBack, robotRotate );
219:
220:        // ******* Test code for JewelKnocker ***********
```

- These lines of code shape and map the joystick inputs for Drive (go) and then give the inputs to go.MoveSimple (the teleop method for Drive)
- Each pass of loop the joystick inputs are shaped (this is what JoystickUtilities is for) and then fed into go to control robot movement
- Lots of commented out code after this was test code and is not described here

End of loop

```
263:        if (gamepad2.right_bumper) { wep.openClaw(); }
264:        if (gamepad2.left_bumper) { wep.closeClaw(); }
265:         // ************* Test code for Claw methods *******
266:
267:         if ( gamepad1.x )
268:         {
269:             claw.claw_Inward();
270:         }
271:
272:         if ( gamepad1.y )
273:         {
274:             claw.claw_Outward();
275:         }
276:
277:
278:         telemetry.update();
279:     }
```

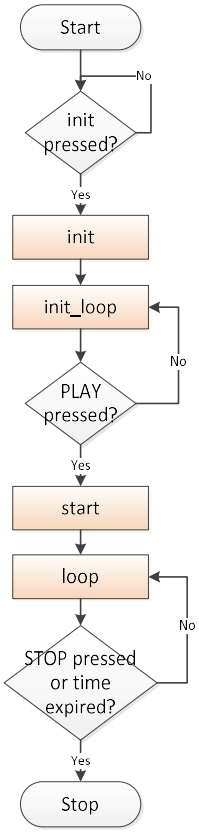
- The last few lines control the claw for glyphs and the claw on the wep for relics
- Finally, the telemetry.update makes sure the screen is updated

Stop and End of Tele_Op_2017

```
281:     /*
282:      * Code to run ONCE after the driver hits STOP
283:      */
284:     @Override
285:     public void stop() {
286:     }
287:
288: }
```

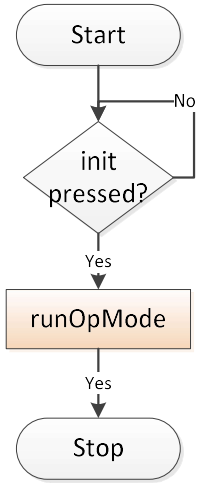
- stop is the last method that must be overridden in an iterative tele op mode
- Again, we didn't need to do anything here, so the method is blank (but you must create it)
- The "}" on line 288 closes out the class

Iterative Op Mode Structure



- As was noted earlier, a iterative op mode has methods that you must override and some of those methods are executed continually until something happens (button press, time out, etc.)
- This flowchart is a drawing of that structure.
- The items in orange boxes are the methods that we must override
- The decision blocks represent the actions that can be taken from the driver control to stop the loops
- All of the tele op modes in the current code follow this structure, and this structure works well for responding to drive inputs and updating multiple outputs

Linear Op Mode Structure



- The other op mode structure is called a Linear OpMode
- This flowchart is a drawing of that structure.
- There is only one method we must override implement in a linear op mode - runOpMode
- The delay to wait for the user to hit "PLAY" and the loops to continue executing must all be built into our code
- All of the competition auto op modes in our code use this structure. This structure makes it easier to take a single action at a time and finish it before moving to the next step.
- The iterative op mode has a time limit on "loop" (around 5 seconds), but there is no time limit on runOpMode

For loop

```
final static int MAX = 10;
int myNum = 25;
for ( int index = 0; index < MAX; index++ )
{
    // Increment myNum each pass of the loop
    myNum++;
}
```

- There were no examples of for loops in the two files, so I have created this simple one as an example
- For loops are used to repeat a block of code multiple times for a fixed count or to iterate through all entries in something like an array
- The loop starts with the keyword “for” and then in parentheses there are three sections separated by ;’s that describe the loop
  - The first section after the for typically creates the “iterator”. In this case, we created an int called index and set it to 0
  - The next section controls how long to run the loop. In this case, the loop will run until index is >= MAX
  - The last section says what to do at the end of each loop, in this case we increment index
- The body of the loop is whatever follows the “for (; ;)” statement. You should always use a { } pair after the for.
- The iterator value (index in this example) can only be used within the body of the for loop
- After this loop completes running, the value of myNum will be 35