```java
// Crater-Claim-Park.java
//--------------------------------------------

// This autonomous OpMode assumes the robot will start hanging on the CRATER side of the alliance's side
// Missions completed: Landing, Sampling
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.util.ElapsedTime;

@Autonomous(name="Crater to Claim", group="Autonomous")
//@Disabled
public class CraterAutoClaimCrator extends LinearOpMode {

    // Declare motors/sensors/members
    private DcMotor leftDrive1 = null;
    private DcMotor rightDrive1 = null;
    private DcMotor intake = null;
    private DcMotor lift = null;
    private ColorSensor colorSensor;
    private ElapsedTime     runtime = new ElapsedTime();

    //Declares variables and constants
    static final double     COUNTS_PER_MOTOR_REV    = 560 ;    //
    static final double     DRIVE_GEAR_REDUCTION    = 1.0 ;     // This is < 1.0 if geared UP
    static final double     WHEEL_DIAMETER_INCHES   = 4.0 ;     // For figuring circumference
    static final double     COUNTS_PER_INCH         = (COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION) /
            (WHEEL_DIAMETER_INCHES * 3.1415);
    static final double     DRIVE_SPEED             = 0.75;
    static final double     TURN_SPEED              = 0.75;

    static final double TURNING_DIAMETER = 18.1; //This and TURNING_CIRCUMFERENCE are used for the turnInPlaceCalc method
    static final double TURNING_CIRCUMFERENCE = TURNING_DIAMETER * 3.1415;

    @Override
    public void runOpMode(){

        // Initialize the hardware variables.
        leftDrive1 = hardwareMap.get(DcMotor.class, "left_drive1"); //First left drive motor
        rightDrive1 = hardwareMap.get(DcMotor.class, "right_drive1"); //First right drive motor
        intake = hardwareMap.get(DcMotor.class, "intake"); //Motor that controls the rubber band intake
        lift = hardwareMap.get(DcMotor.class, "lift1"); //Motor that controls the lift
        colorSensor = hardwareMap.colorSensor.get("color"); //Color sensor for sampling

        // Sets direction of motors
        leftDrive1.setDirection(DcMotor.Direction.REVERSE); //Left drive is reversed
        rightDrive1.setDirection(DcMotor.Direction.FORWARD);

        // Send telemetry message to signify robot waiting
        telemetry.addData("Status", "Waiting");
        telemetry.update();
        // Resets Encoders [no longer in use]
        //rightDrive1.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        // Send telemetry message to indicate successful Encoder reset
        //telemetry.addData("Path0", "Starting at %7d",
        //      rightDrive1.getCurrentPosition());
        //telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();
        colorSensor.enableLed(true);

        // Note: When using encoderDrive(), reverse movement is obtained by setting a negative distance (not speed)

        //Extends arm; completes Landing mission
        lift.setPower(-0.75);
        sleep(2800);
        lift.setPower(0.0);


        msDrive(-0.5, -0.5, 250); //Backs away from hook on Lander
        msDrive(0.5, -0.5, 900); //Turns to depot

        msDrive(0.35, 0.68, 2150); //Wide arc-turn toward depot
        msDrive(0.75,0.75,1375); //Move forward into depot

        //Releases team marker and Claims Depot using intake rollers
```

```
        intake.setPower(0.6);
        msDrive(-0.3,-0.3,550);
        intake.setPower(0.0);

        //Backing up to crater REMOVED for this OpMode; see Crater-Claim-Park.java

    }
    /*
     *  Method to perform a relative move, based on encoder counts.
     *  Encoders are not reset as the move is based on the current position.
     *  Move will stop if any of three conditions occur:
     *  1) Move gets to the desired position
     *  2) Move runs out of time
     *  3) Driver stops the opmode running.
     */
    public void encoderDrive(double speed,
                             double inches,
                             double timeoutS) {
        // Defines targets for both motors  EDIT: Left encoder not funcional; rewriting to incorporate only one
        int newRightTarget;
        // Ensure that the opmode is still active
        if (opModeIsActive()) {
            // Determine new target position for each motor, and pass to motor controller
            newRightTarget = rightDrive1.getCurrentPosition() + (int)(inches * COUNTS_PER_INCH);
            rightDrive1.setMode(DcMotor.RunMode.RUN_TO_POSITION);
            rightDrive1.setTargetPosition(newRightTarget);

            // Set mode for "2" drives if it doesn't work 11/20/18
            // Resets timeout and starts motion
            runtime.reset();
            leftDrive1.setPower(Math.abs(speed));
            rightDrive1.setPower(Math.abs(speed));
            telemetry.addData("spot1", rightDrive1.isBusy());

            // keeps looping while we are still active, and there is time left, and both motors are running.
            while (opModeIsActive() && rightDrive1.isBusy() && runtime.seconds() < timeoutS)
            {
                telemetry.addData("Path1",  "Running to %d ",  newRightTarget);
                telemetry.addData("Path2",  "Running at %d ", rightDrive1.getCurrentPosition());
                telemetry.update();
            }
            // Stop all motion;
            leftDrive1.setPower(0);
            rightDrive1.setPower(0);

            // Turn off RUN_TO_POSITION
            rightDrive1.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
        }
    }

    //Function that does the calculations for turning in place; to be used with encoderDrive
    public double turnInPlaceCalc(int degrees){
        return ((degrees / 360) * TURNING_CIRCUMFERENCE);
    }

    //Function for making the robot move at set left and right speeds for a set amount of time (ms)
    public void msDrive(double leftSpeed, double rightSpeed, long ms) {
        leftDrive1.setPower(leftSpeed);
        rightDrive1.setPower(rightSpeed);
        sleep(ms);
        leftDrive1.setPower(0.0);
        rightDrive1.setPower(0.0);
    }

    //Function that uses the color sensor to test for gold color (Sampling)
    //NOW FUNCTIONAL! Note: needs to be implement
    public boolean testIfGold() {
        float red = (float)colorSensor.red();
        float green = (float)colorSensor.green();
        float blue = (float)colorSensor.blue();
        return (((red / blue) > 1.5) && ((red / blue) < 3.2) && ((blue / green) > 0.37) && ((blue / green) < 0.68));
        /* Testing for a range of values does not work because red, blue and green, change drastically depending on the
        *  distance between the color sensor and the mineral being tested. However the ratio of red to blue to green is
        *  always constant for the same hue of gold. Therefore checking for the right range will work. Ranges were
        *  calculated by gathering the RGB readings at multiple distancing and finding the average ratios. */
    }
}
```

```java
// Crater-Claim.java
//--------------------------------------------

// This autonomous OpMode assumes the robot will start hanging on the CRATER side of the alliance's side
// Missions completed: Landing, Sampling
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.util.ElapsedTime;

@Autonomous(name="Crater to Claim", group="Autonomous")
//@Disabled
public class CraterAutoClaimCrator extends LinearOpMode {

    // Declare motors/sensors/members
    private DcMotor leftDrive1 = null;
    private DcMotor rightDrive1 = null;
    private DcMotor intake = null;
    private DcMotor lift = null;
    private ColorSensor colorSensor;
    private ElapsedTime     runtime = new ElapsedTime();

    //Declares variables and constants
    static final double     COUNTS_PER_MOTOR_REV    = 560 ;    //
    static final double     DRIVE_GEAR_REDUCTION    = 1.0 ;     // This is < 1.0 if geared UP
    static final double     WHEEL_DIAMETER_INCHES   = 4.0 ;     // For figuring circumference
    static final double     COUNTS_PER_INCH         = (COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION) /
            (WHEEL_DIAMETER_INCHES * 3.1415);
    static final double     DRIVE_SPEED             = 0.75;
    static final double     TURN_SPEED              = 0.75;

    static final double TURNING_DIAMETER = 18.1; //This and TURNING_CIRCUMFERENCE are used for the turnInPlaceCalc method
    static final double TURNING_CIRCUMFERENCE = TURNING_DIAMETER * 3.1415;

    @Override
    public void runOpMode(){

        // Initialize the hardware variables.
        leftDrive1 = hardwareMap.get(DcMotor.class, "left_drive1"); //First left drive motor
        rightDrive1 = hardwareMap.get(DcMotor.class, "right_drive1"); //First right drive motor
        intake = hardwareMap.get(DcMotor.class, "intake"); //Motor that controls the rubber band intake
        lift = hardwareMap.get(DcMotor.class, "lift1"); //Motor that controls the lift
        colorSensor = hardwareMap.colorSensor.get("color"); //Color sensor for sampling

        // Sets direction of motors
        leftDrive1.setDirection(DcMotor.Direction.REVERSE); //Left drive is reversed
        rightDrive1.setDirection(DcMotor.Direction.FORWARD);

        // Send telemetry message to signify robot waiting
        telemetry.addData("Status", "Waiting");
        telemetry.update();
        // Resets Encoders [no longer in use]
        //rightDrive1.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        // Send telemetry message to indicate successful Encoder reset
        //telemetry.addData("Path0", "Starting at %7d",
        //      rightDrive1.getCurrentPosition());
        //telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();
        colorSensor.enableLed(true);

        // Note: When using encoderDrive(), reverse movement is obtained by setting a negative distance (not speed)

        //Extends arm; completes Landing mission
        lift.setPower(-0.75);
        sleep(2800);
        lift.setPower(0.0);


        msDrive(-0.5, -0.5, 250); //Backs away from hook on Lander
        msDrive(0.5, -0.5, 900); //Turns to depot

        msDrive(0.35, 0.68, 2150); //Wide arc-turn toward depot
        msDrive(0.75,0.75,1375); //Move forward into depot

        //Releases team marker and Claims Depot using intake rollers
```

```
        intake.setPower(0.6);
        msDrive(-0.3,-0.3,550);
        intake.setPower(0.0);

        //Backing up to crater REMOVED for this OpMode; see Crater-Claim-Park.java

    }
    /*
     *  Method to perform a relative move, based on encoder counts.
     *  Encoders are not reset as the move is based on the current position.
     *  Move will stop if any of three conditions occur:
     *  1) Move gets to the desired position
     *  2) Move runs out of time
     *  3) Driver stops the opmode running.
     */
    public void encoderDrive(double speed,
                             double inches,
                             double timeoutS) {
        // Defines targets for both motors  EDIT: Left encoder not funcional; rewriting to incorporate only one
        int newRightTarget;
        // Ensure that the opmode is still active
        if (opModeIsActive()) {
            // Determine new target position for each motor, and pass to motor controller
            newRightTarget = rightDrive1.getCurrentPosition() + (int)(inches * COUNTS_PER_INCH);
            rightDrive1.setMode(DcMotor.RunMode.RUN_TO_POSITION);
            rightDrive1.setTargetPosition(newRightTarget);

            // Set mode for "2" drives if it doesn't work 11/20/18
            // Resets timeout and starts motion
            runtime.reset();
            leftDrive1.setPower(Math.abs(speed));
            rightDrive1.setPower(Math.abs(speed));
            telemetry.addData("spot1", rightDrive1.isBusy());

            // keeps looping while we are still active, and there is time left, and both motors are running.
            while (opModeIsActive() && rightDrive1.isBusy() && runtime.seconds() < timeoutS)
            {
                telemetry.addData("Path1",  "Running to %d ",  newRightTarget);
                telemetry.addData("Path2",  "Running at %d ", rightDrive1.getCurrentPosition());
                telemetry.update();
            }
            // Stop all motion;
            leftDrive1.setPower(0);
            rightDrive1.setPower(0);

            // Turn off RUN_TO_POSITION
            rightDrive1.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
        }
    }

    //Function that does the calculations for turning in place; to be used with encoderDrive
    public double turnInPlaceCalc(int degrees){
        return ((degrees / 360) * TURNING_CIRCUMFERENCE);
    }

    //Function for making the robot move at set left and right speeds for a set amount of time (ms)
    public void msDrive(double leftSpeed, double rightSpeed, long ms) {
        leftDrive1.setPower(leftSpeed);
        rightDrive1.setPower(rightSpeed);
        sleep(ms);
        leftDrive1.setPower(0.0);
        rightDrive1.setPower(0.0);
    }

    //Function that uses the color sensor to test for gold color (Sampling)
    //NOW FUNCTIONAL! Note: needs to be implement
    public boolean testIfGold() {
        float red = (float)colorSensor.red();
        float green = (float)colorSensor.green();
        float blue = (float)colorSensor.blue();
        return (((red / blue) > 1.5) && ((red / blue) < 3.2) && ((blue / green) > 0.37) && ((blue / green) < 0.68));
        /* Testing for a range of values does not work because red, blue and green, change drastically depending on the
        *  distance between the color sensor and the mineral being tested. However the ratio of red to blue to green is
        *  always constant for the same hue of gold. Therefore checking for the right range will work. Ranges were
        *  calculated by gathering the RGB readings at multiple distancing and finding the average ratios. */
    }
}
```

```java
// Crater-Park.java
//--------------------------------------------

// This autonomous OpMode assumes the robot will start hanging on the CRATER side of the alliance's side
// Missions completed: Landing, Parking
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.util.ElapsedTime;

@Autonomous(name="MAIN Crater to Claim to Park", group="Autonomous")
//@Disabled
public class CraterAutoClaimCrator extends LinearOpMode {

    // Declare motors/sensors/members
    private DcMotor leftDrive1 = null;
    private DcMotor rightDrive1 = null;
    private DcMotor intake = null;
    private DcMotor lift = null;
    private ColorSensor colorSensor;
    private ElapsedTime     runtime = new ElapsedTime();

    //Declares variables and constants
    static final double     COUNTS_PER_MOTOR_REV    = 560 ;    //
    static final double     DRIVE_GEAR_REDUCTION    = 1.0 ;     // This is < 1.0 if geared UP
    static final double     WHEEL_DIAMETER_INCHES   = 4.0 ;     // For figuring circumference
    static final double     COUNTS_PER_INCH         = (COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION) /
            (WHEEL_DIAMETER_INCHES * 3.1415);
    static final double     DRIVE_SPEED             = 0.75;
    static final double     TURN_SPEED              = 0.75;

    static final double TURNING_DIAMETER = 18.1; //This and TURNING_CIRCUMFERENCE are used for the turnInPlaceCalc method
    static final double TURNING_CIRCUMFERENCE = TURNING_DIAMETER * 3.1415;

    @Override
    public void runOpMode(){

        // Initialize the hardware variables.
        leftDrive1 = hardwareMap.get(DcMotor.class, "left_drive1"); //First left drive motor
        rightDrive1 = hardwareMap.get(DcMotor.class, "right_drive1"); //First right drive motor
        intake = hardwareMap.get(DcMotor.class, "intake"); //Motor that controls the rubber band intake
        lift = hardwareMap.get(DcMotor.class, "lift1"); //Motor that controls the lift
        colorSensor = hardwareMap.colorSensor.get("color"); //Color sensor for sampling

        // Sets direction of motors
        leftDrive1.setDirection(DcMotor.Direction.REVERSE); //Left drive is reversed
        rightDrive1.setDirection(DcMotor.Direction.FORWARD);

        // Send telemetry message to signify robot waiting
        telemetry.addData("Status", "Waiting");
        telemetry.update();
        // Resets Encoders [no longer in use]
        //rightDrive1.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        // Send telemetry message to indicate successful Encoder reset
        //telemetry.addData("Path0", "Starting at %7d",
        //      rightDrive1.getCurrentPosition());
        //telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();
        colorSensor.enableLed(true);

        // Note: When using encoderDrive(), reverse movement is obtained by setting a negative distance (not speed)

        //Extends arm; completes Landing mission
        lift.setPower(-0.75);
        sleep(2800);
        lift.setPower(0.0);


        msDrive(-0.5, -0.5, 250); //Backs away from hook on Lander
        msDrive(0.5, -0.5, 800); //Turns to crater

        // Skips Depot Step; see Crater-Claim-Park.java

        //Drives to Crater
        msDrive(-0.9, -0.9, 2675);
```

```
    }
    /*
     *  Method to perform a relative move, based on encoder counts.
     *  Encoders are not reset as the move is based on the current position.
     *  Move will stop if any of three conditions occur:
     *  1) Move gets to the desired position
     *  2) Move runs out of time
     *  3) Driver stops the opmode running.
     */
    public void encoderDrive(double speed,
                             double inches,
                             double timeoutS) {
        // Defines targets for both motors  EDIT: Left encoder not funcional; rewriting to incorporate only one
        int newRightTarget;
        // Ensure that the opmode is still active
        if (opModeIsActive()) {
            // Determine new target position for each motor, and pass to motor controller
            newRightTarget = rightDrive1.getCurrentPosition() + (int)(inches * COUNTS_PER_INCH);
            rightDrive1.setMode(DcMotor.RunMode.RUN_TO_POSITION);
            rightDrive1.setTargetPosition(newRightTarget);

            // Set mode for "2" drives if it doesn't work 11/20/18
            // Resets timeout and starts motion
            runtime.reset();
            leftDrive1.setPower(Math.abs(speed));
            rightDrive1.setPower(Math.abs(speed));
            telemetry.addData("spot1", rightDrive1.isBusy());

            // keeps looping while we are still active, and there is time left, and both motors are running.
            while (opModeIsActive() && rightDrive1.isBusy() && runtime.seconds() < timeoutS)
            {
                telemetry.addData("Path1",  "Running to %d ",  newRightTarget);
                telemetry.addData("Path2",  "Running at %d ", rightDrive1.getCurrentPosition());
                telemetry.update();
            }
            // Stop all motion;
            leftDrive1.setPower(0);
            rightDrive1.setPower(0);

            // Turn off RUN_TO_POSITION
            rightDrive1.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
        }
    }

    //Function that does the calculations for turning in place; to be used with encoderDrive
    public double turnInPlaceCalc(int degrees){
        return ((degrees / 360) * TURNING_CIRCUMFERENCE);
    }

    //Function for making the robot move at set left and right speeds for a set amount of time (ms)
    public void msDrive(double leftSpeed, double rightSpeed, long ms) {
        leftDrive1.setPower(leftSpeed);
        rightDrive1.setPower(rightSpeed);
        sleep(ms);
        leftDrive1.setPower(0.0);
        rightDrive1.setPower(0.0);
    }

    //Function that uses the color sensor to test for gold color (Sampling)
    //NOW FUNCTIONAL! Note: needs to be implement
    public boolean testIfGold() {
        float red = (float)colorSensor.red();
        float green = (float)colorSensor.green();
        float blue = (float)colorSensor.blue();
        return (((red / blue) > 1.5) && ((red / blue) < 3.2) && ((blue / green) > 0.37) && ((blue / green) < 0.68));
        /* Testing for a range of values does not work because red, blue and green, change drastically depending on the
        *  distance between the color sensor and the mineral being tested. However the ratio of red to blue to green is
        *  always constant for the same hue of gold. Therefore checking for the right range will work. Ranges were
        *  calculated by gathering the RGB readings at multiple distancing and finding the average ratios. */
    }
}
```

```
// NO-LIFT.java
//--------------------------------------------

// This autonomous OpMode assumes the robot will start ON THE GROUND on EITHER SIDE of the lander.
// This program can be used for TWO PURPOSES.
//      EITHER: Starting on the Depot side and Claiming the Depot   OR   Starting on the Crater side and Parking on the Crater
// Missions completed: Claiming OR Parking
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.util.ElapsedTime;

@Autonomous(name="NO LIFT Depot-Claim OR Crater-Park", group="Autonomous")
//@Disabled
public class CraterAutoClaimCrator extends LinearOpMode {

    // Declare motors/sensors/members
    private DcMotor leftDrive1 = null;
    private DcMotor rightDrive1 = null;
    private DcMotor intake = null;
    private DcMotor lift = null;
    private ColorSensor colorSensor;
    private ElapsedTime     runtime = new ElapsedTime();

    //Declares variables and constants
    static final double     COUNTS_PER_MOTOR_REV    = 560 ;    //
    static final double     DRIVE_GEAR_REDUCTION    = 1.0 ;     // This is < 1.0 if geared UP
    static final double     WHEEL_DIAMETER_INCHES   = 4.0 ;     // For figuring circumference
    static final double     COUNTS_PER_INCH         = (COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION) /
            (WHEEL_DIAMETER_INCHES * 3.1415);
    static final double     DRIVE_SPEED             = 0.75;
    static final double     TURN_SPEED              = 0.75;

    static final double TURNING_DIAMETER = 18.1; //This and TURNING_CIRCUMFERENCE are used for the turnInPlaceCalc method
    static final double TURNING_CIRCUMFERENCE = TURNING_DIAMETER * 3.1415;

    @Override
    public void runOpMode(){

        // Initialize the hardware variables.
        leftDrive1 = hardwareMap.get(DcMotor.class, "left_drive1"); //First left drive motor
        rightDrive1 = hardwareMap.get(DcMotor.class, "right_drive1"); //First right drive motor
        intake = hardwareMap.get(DcMotor.class, "intake"); //Motor that controls the rubber band intake
        lift = hardwareMap.get(DcMotor.class, "lift1"); //Motor that controls the lift
        colorSensor = hardwareMap.colorSensor.get("color"); //Color sensor for sampling

        // Sets direction of motors
        leftDrive1.setDirection(DcMotor.Direction.REVERSE); //Left drive is reversed
        rightDrive1.setDirection(DcMotor.Direction.FORWARD);

        // Send telemetry message to signify robot waiting
        telemetry.addData("Status", "Waiting");
        telemetry.update();
        // Resets Encoders [no longer in use]
        //rightDrive1.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        // Send telemetry message to indicate successful Encoder reset
        //telemetry.addData("Path0", "Starting at %7d",
        //      rightDrive1.getCurrentPosition());
        //telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();
        colorSensor.enableLed(true);

        // Note: When using encoderDrive(), reverse movement is obtained by setting a negative distance (not speed)

        msDrive(0.75,0.75,750); //Move forward into Depot OR Crater

        //Releases team marker and Claims Depot using intake rollers (or spins intake and does nothing, if on crater side)
        intake.setPower(0.6);
        msDrive(-0.3,-0.3,550);
        intake.setPower(0.0);

        //Backing up to crater REMOVED for this OpMode; see Crater-Claim-Park.java

    }
    /*
```

```
     *  Method to perform a relative move, based on encoder counts.
     *  Encoders are not reset as the move is based on the current position.
     *  Move will stop if any of three conditions occur:
     *  1) Move gets to the desired position
     *  2) Move runs out of time
     *  3) Driver stops the opmode running.
     */
    public void encoderDrive(double speed,
                             double inches,
                             double timeoutS) {
        // Defines targets for both motors  EDIT: Left encoder not funcional; rewriting to incorporate only one
        int newRightTarget;
        // Ensure that the opmode is still active
        if (opModeIsActive()) {
            // Determine new target position for each motor, and pass to motor controller
            newRightTarget = rightDrive1.getCurrentPosition() + (int)(inches * COUNTS_PER_INCH);
            rightDrive1.setMode(DcMotor.RunMode.RUN_TO_POSITION);
            rightDrive1.setTargetPosition(newRightTarget);

            // Set mode for "2" drives if it doesn't work 11/20/18
            // Resets timeout and starts motion
            runtime.reset();
            leftDrive1.setPower(Math.abs(speed));
            rightDrive1.setPower(Math.abs(speed));
            telemetry.addData("spot1", rightDrive1.isBusy());

            // keeps looping while we are still active, and there is time left, and both motors are running.
            while (opModeIsActive() && rightDrive1.isBusy() && runtime.seconds() < timeoutS)
            {
                telemetry.addData("Path1",  "Running to %d ",  newRightTarget);
                telemetry.addData("Path2",  "Running at %d ", rightDrive1.getCurrentPosition());
                telemetry.update();
            }
            // Stop all motion;
            leftDrive1.setPower(0);
            rightDrive1.setPower(0);

            // Turn off RUN_TO_POSITION
            rightDrive1.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
        }
    }

    //Function that does the calculations for turning in place; to be used with encoderDrive
    public double turnInPlaceCalc(int degrees){
        return ((degrees / 360) * TURNING_CIRCUMFERENCE);
    }

    //Function for making the robot move at set left and right speeds for a set amount of time (ms)
    public void msDrive(double leftSpeed, double rightSpeed, long ms) {
        leftDrive1.setPower(leftSpeed);
        rightDrive1.setPower(rightSpeed);
        sleep(ms);
        leftDrive1.setPower(0.0);
        rightDrive1.setPower(0.0);
    }

    //Function that uses the color sensor to test for gold color (Sampling)
    //NOW FUNCTIONAL! Note: needs to be implement
    public boolean testIfGold() {
        float red = (float)colorSensor.red();
        float green = (float)colorSensor.green();
        float blue = (float)colorSensor.blue();
        return (((red / blue) > 1.5) && ((red / blue) < 3.2) && ((blue / green) > 0.37) && ((blue / green) < 0.68));
        /* Testing for a range of values does not work because red, blue and green, change drastically depending on the
        *  distance between the color sensor and the mineral being tested. However the ratio of red to blue to green is
        *  always constant for the same hue of gold. Therefore checking for the right range will work. Ranges were
        *  calculated by gathering the RGB readings at multiple distancing and finding the average ratios. */
    }
}
```

```java
// MainTeleop.java
//---------------------------------------------

// This OpMode is for the Driver-Controlled Period
/* Controls:
*     Controller 1:
*       Left Stick = Left Drive; Right Stick = Right Drive
*     Controller 2:
*       D-Pad Up/Down = Lift Up/Down
*       Y [Macro] = Lift Completely Up; A [Macro] = Lift Completely Down **WARNING: DO NOT USE Y/A IF LIFT IS NOT EITHER
COMPLETELY
*                                                                                    UP/DOWN TO AVOID GRINDING THE GEARS**
*       Left Trigger = Intake/Outtake Rollers IN; Right Trigger = Intake/Outtake Rollers OUT (Outtake)
*/
package org.firstinspires.ftc.teamcode;

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.util.ElapsedTime;

@TeleOp(name="MainTeleOp", group="Linear Opmode")
//@Disabled
public class MainTeleop extends LinearOpMode {
    // Declare motors/sensors/members
    private ElapsedTime runtime = new ElapsedTime();
    private DcMotor leftDrive1 = null;
    private DcMotor rightDrive1 = null;
    private DcMotor intake = null;
    private DcMotor lift = null;
    private ColorSensor colorSensor;

    @Override
    public void runOpMode() {
        telemetry.addData("Status", "Initialized");
        telemetry.update();

        // Initialize the hardware variables.
        leftDrive1 = hardwareMap.get(DcMotor.class, "left_drive1");
        rightDrive1 = hardwareMap.get(DcMotor.class, "right_drive1");
        intake = hardwareMap.get(DcMotor.class, "intake");
        lift = hardwareMap.get(DcMotor.class, "lift1");
        colorSensor = hardwareMap.colorSensor.get("color");

        // Sets directions of motors.
        rightDrive1.setDirection(DcMotor.Direction.REVERSE);
        leftDrive1.setDirection(DcMotor.Direction.FORWARD);
        // Wait for the game to start (driver presses PLAY)
        waitForStart();
        runtime.reset();
        // Enables color sensor LED [not used]
        // colorSensor.enableLed(true);
        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {
            // Setup a variable for each drive wheel to save power level for telemetry
            double leftPower;
            double rightPower;

            // Map power to joysticks
            leftPower = gamepad1.left_stick_y;
            rightPower = gamepad1.right_stick_y;

            // Send power to wheels
            leftDrive1.setPower(leftPower);
            rightDrive1.setPower(rightPower);

            // Maps intake/outtake rollers to triggers
            if (gamepad2.left_trigger != 0) {
                intake.setPower(0.5);
            } else if (gamepad2.right_trigger != 0) {
                intake.setPower(-0.5);
            } else {
                intake.setPower(0.0);
            }

            // Maps lift to dpad
```

```
            if (gamepad2.dpad_up) {
                lift.setPower(-0.6);
            } else if (gamepad2.dpad_down) {
                lift.setPower(0.6);
            } else {
                lift.setPower(0.0);
            }

            // Lift Completely Up/Down Macros
            if (gamepad2.y) {
                lift.setPower(-0.6);
                sleep(4400);
                lift.setPower(0.0);
            }
            if (gamepad2.a) {
                lift.setPower(0.6);
                sleep(4400);
                lift.setPower(0.0);
            }

            // Show the elapsed game time and wheel power.
            telemetry.addData("Status", "Run Time: " + runtime.toString());
            telemetry.addData("Motors", "left (%.2f), right (%.2f)", leftPower, rightPower);
            telemetry.addData("Motor Encoders", "encoder: %d %d", leftDrive1.getCurrentPosition(),
rightDrive1.getCurrentPosition());
            //telemetry.addData("ColorSensor", "colo red %d blue %d green %d ", colorSensor.red(), colorSensor.blue(),
colorSensor.green());
            telemetry.update();
        }
    }
}
```

```
// Various-Code-Snippets.java
// All of our code can be found in the Engineering Notebook. These are some snippets that are associated with the Control
Award and are therefore presented here.
// ------------------------------------------------------------

// encoderDrive
    /*
     *  Method to perform a relative move, based on encoder counts.
     *  Encoders are not reset as the move is based on the current position.
     *  Move will stop if any of three conditions occur:
     *  1) Move gets to the desired position
     *  2) Move runs out of time
     *  3) Driver stops the opmode running.
     */
public void encoderDrive(double speed,
                         double inches,
                         double timeoutS) {
        // Defines targets for both motors  EDIT: Left encoder not funcional; rewriting to incorporate only one
        int newRightTarget;
        // Ensure that the opmode is still active
        if (opModeIsActive()) {
            // Determine new target position for each motor, and pass to motor controller
            newRightTarget = rightDrive1.getCurrentPosition() + (int)(inches * COUNTS_PER_INCH);
            rightDrive1.setMode(DcMotor.RunMode.RUN_TO_POSITION);
            rightDrive1.setTargetPosition(newRightTarget);

            // Set mode for "2" drives if it doesn't work 11/20/18
            // Resets timeout and starts motion
            runtime.reset();
            leftDrive1.setPower(Math.abs(speed));
            rightDrive1.setPower(Math.abs(speed));
            telemetry.addData("spot1", rightDrive1.isBusy());

            // keeps looping while we are still active, and there is time left, and both motors are running.
            while (opModeIsActive() && rightDrive1.isBusy() && runtime.seconds() < timeoutS)
            {
                telemetry.addData("Path1",  "Running to %d ",  newRightTarget);
                telemetry.addData("Path2",  "Running at %d ", rightDrive1.getCurrentPosition());
                telemetry.update();
            }
            // Stop all motion;
            leftDrive1.setPower(0);
            rightDrive1.setPower(0);

            // Turn off RUN_TO_POSITION
            rightDrive1.setMode(DcMotor.RunMode.RUN_USING_ENCODER);
        }
    }
// turnInPlaceCalc
// Function that does the calculations for turning in place; to be used with encoderDrive
    public double turnInPlaceCalc(int degrees){
        return ((degrees / 360) * TURNING_CIRCUMFERENCE);
    }
// msDrive
//Function for making the robot move at set left and right speeds for a set amount of time (ms)
    public void msDrive(double leftSpeed, double rightSpeed, long ms) {
        leftDrive1.setPower(leftSpeed);
        rightDrive1.setPower(rightSpeed);
        sleep(ms);
        leftDrive1.setPower(0.0);
        rightDrive1.setPower(0.0);
    }

// testIfGold
//Function that uses the color sensor to test for gold color (Sampling)
    //NOW FUNCTIONAL! Note: needs to be implement
    public boolean testIfGold() {
        float red = (float)colorSensor.red();
        float green = (float)colorSensor.green();
        float blue = (float)colorSensor.blue();
        return (((red / blue) > 1.5) && ((red / blue) < 3.2) && ((blue / green) > 0.37) && ((blue / green) < 0.68));
        /* Testing for a range of values does not work because red, blue and green, change drastically depending on the
        *  distance between the color sensor and the mineral being tested. However the ratio of red to blue to green is
        *  always constant for the same hue of gold. Therefore checking for the right range will work. Ranges were
        *  calculated by gathering the RGB readings at multiple distancing and finding the average ratios. */
    }
// Thank you for your time
// FTC 4390 Storm Robotics Typhoons
```