Olá robô - Controle do Robô (OnBot Java)

- Controle do robô
- Navegação do robô - OnBot Java
- Tempo Decorrido - OnBot Java
- Controle de braço - OnBot Java
- Navegação por Encoder

Controle do robô

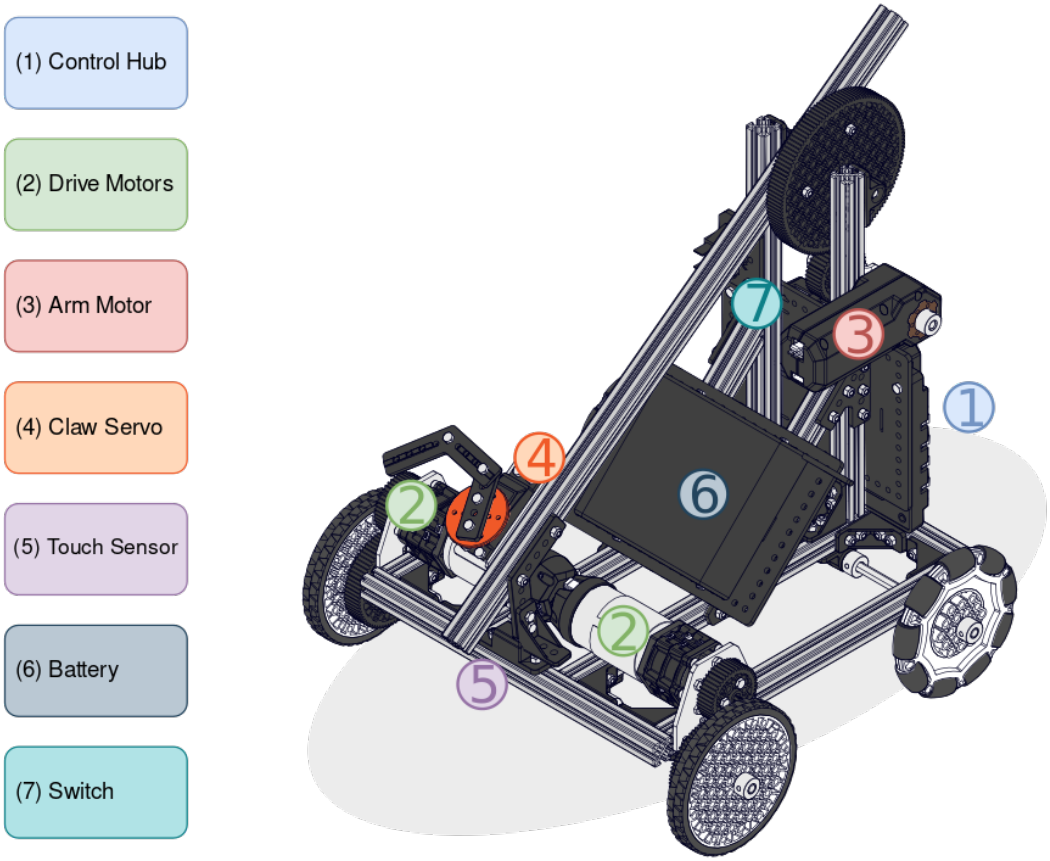
Compreendidos os princípios básicos de controle de atuadores e obtenção de feedback de sensores a partir do Olá Robô - Test Bed, é hora de começar a configurar e programar nosso robô para controle Teleoperado e Autônomo!

Seção	Objetivos da seção
Criando um robô básico	Apresenta um potencial robô para trabalhar, assim como o arquivo de configuração utilizado nas seções seguintes.
Noções básicas de transmissão	Diferenças entre drivetrains diferencial e omnidirecional e seu impacto nos tipos de controle teleoperado.

> Antes de continuar, é recomendado completar, no mínimo, um drivetrain. Existem algumas opções diferentes dependendo do kit que está sendo utilizado.
>
> Para este guia, o Class Bot V2 é utilizado. Consulte o guia de montagem para obter instruções completas de montagem para o Class Bot V2!

Criar um robô básico

A imagem abaixo destaca os principais componentes de hardware do Class Bot V2. Esses componentes são importantes para entender o processo de configuração.



A seção de Configuração do Olá Robô concentrou-se na configuração dos componentes no Test Bed. Para avançar nas seções de programação do Controle do Robô, é necessário criar um novo arquivo de configuração para os componentes no robô. É sua escolha quais nomes de variáveis você deseja atribuir ao seu robô, mas, para referência, este guia usará os seguintes nomes para cada componente de hardware.

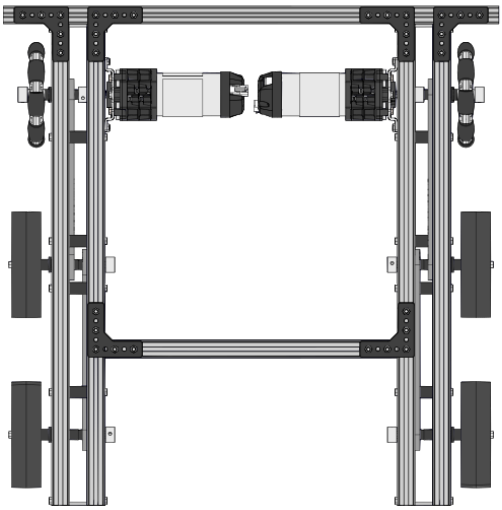
Componente de Hardware	Tipo de Hardware	Nome
Motor direito	REV Robotics UltraPlanetary HD Hex Motor	right motor
Motor esquerdo	REV Robotics UltraPlanetary HD Hex Motor	left motor
Braço motor	REV Robotics Core Hex Motor	arm
Garra servo	Servo	claw
Sensor de toque	REV Touch Sensor	touch

Noções básicas de transmissão

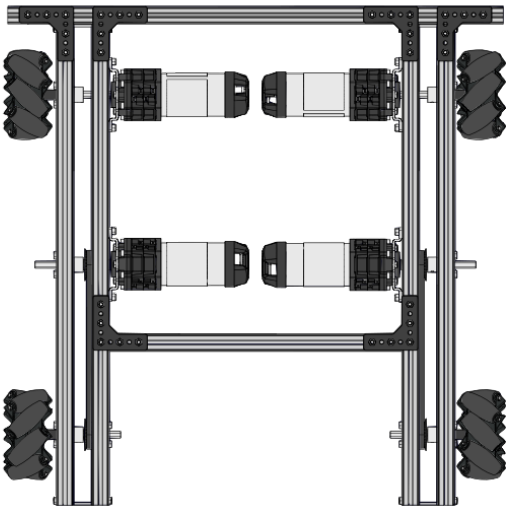
Antes de continuar, é importante entender o comportamento mecânico de diferentes drivetrains. As duas categorias mais comuns de drivetrains são Diferencial e Omnidirecional. O drivetrain do Class Bot é um drivetrain diferencial. A tabela abaixo destaca as principais características desses

dois tipos de drivetrains.

Differential



Omnidirectional



Tração diferencial	Omnidirecional
Tipo mais comum de transmissão	Pode se mover em qualquer direção
Se move ao longo de um eixo central	Varia a força em cada roda para se mover lateralmente ou linearmente
Aplica mais potência de um dos lados para para mudar a direção	Programação mais complexa
Pode ter diferentes nomes (4WD, 6WD, West Coast..)	Precisa de mais de dois motores

Tipos de controle teleoperado

Existem várias maneiras de controlar um robô teleoperado. Ao usar o REV Control System, isso é feito com um dispositivo de Driver Station e gamepads. Existem várias maneiras de usar um controlador para movimentar um drivetrain diferencial. Duas das formas convencionais são Tank Drive e Arcade Drive.

Tração Tank

Para o Tank Drive, cada lado do drivetrain diferencial é mapeado para seu próprio joystick. Alterar a posição de cada joystick permite que o drivetrain vire e mude sua direção. Existe um código de exemplo no aplicativo Controlador do Robô para controlar um drivetrain diferencial dessa maneira.

Arcade Drive

Para o Arcade Drive, cada lado do drivetrain diferencial é controlado por um único joystick. Alterar a posição do joystick muda a potência aplicada a cada lado do drivetrain, permitindo um comando específico. Os controles de Arcade Drive geralmente têm o movimento esquerda/direita do joystick configurado para girar o robô em torno do seu eixo, com o movimento para frente/para trás fazendo o robô avançar e retroceder. Mais informações sobre Arcade Drive podem ser encontradas nas próximas seções.

Com o robô configurado e uma compreensão básica de drivetrains e tipos de controle teleoperado, podemos avançar para a programação do drivetrain para movimentar o robô.

Navegação do robô - OnBot Java

Introdução a navegação do robô

Como indicado na seção Olá Robô - Controle do Robô, o controle de robôs assume muitas formas diferentes. Um dos tipos de controle a ser considerado para robôs com drivetrains é a navegação do robô.

A navegação do robô como conceito depende do tipo de drivetrain e do tipo de modo de operação. Por exemplo, o código para controlar um drivetrain mecanum difere do código usado para controlar um drivetrain diferencial. Também há diferença entre codificar para direção teleoperada, com um gamepad, e codificar para autonomia, onde cada movimento do robô deve ser definido dentro do código.

A próxima seção passa pelos fundamentos da programação para um drivetrain diferencial, bem como como configurar um código de drivetrain teleoperado no estilo arcade. Os conceitos e a lógica destacados nesta seção serão aplicáveis à seção de controle autônomo Elapsed Time.

Seções	Objetivos da seção
Noções básicas de programação de transmissão	O que considerar quando estiver programando uma transmissão e como aplicar.

Noções básicas de programação de transmissão

Programação de motores de transmissão

Comece criando um Op Mode chamado de DualDrive.

> Visite a seção OnBot Java para obter mais informações sobre como criar um modo operacional (op mode). O op mode abaixo concentra-se apenas no mapeamento de hardware dos motores relevantes do sistema de transmissão.

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

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;

@TeleOp
public class DualDrive extends LinearOpMode {
  private DcMotor rightmotor;
  private DcMotor leftmotor;

  @Override
  public void runOpMode() {

        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");

        waitForStart();

        while (opModeIsActive()) {

        }
    }
}
```

Dado que o foco desta seção é criar um sistema de transmissão funcional no código, vamos começar adicionando rightmotor.setPower(1); e leftmotor.setPower(1); ao loop while do op mode.

```
while (opModeIsActive()) {
    rightmotor.setPower(1);
    leftmotor.setPower(1);
    }
```

> Antes de prosseguir, tente executar o código conforme está e considere as seguintes perguntas:
>
> - Qual comportamento o robô está exibindo?
> - Em que direção o robô está girando?

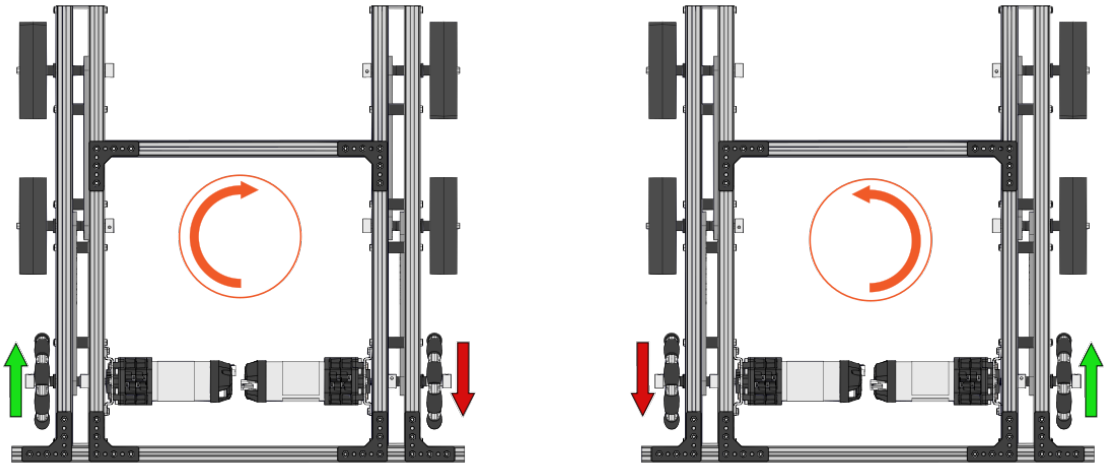
> Quando os motores funcionam em velocidades diferentes, eles giram em torno do ponto central de pivô. No entanto, os motores estão ambos configurados com uma potência (ou ciclo de trabalho) de 1?

Os motores de corrente contínua são capazes de girar em duas direções diferentes, dependendo do fluxo de corrente: no sentido horário e no sentido anti-horário. Ao usar um valor de potência positivo, o Control Hub envia corrente para o motor para fazê-lo girar no sentido horário. Com a classe Bot e o código atual, ambos os motores estão atualmente configurados para girar no sentido horário. No entanto, se você colocar o robô em blocos e executar o código novamente, verá que os motores giram em direções opostas. Com a maneira espelhada como os motores são montados no sistema de transmissão, um motor é naturalmente o inverso do outro. Por que o motor inverso faz o robô girar em círculos? Tanto a velocidade quanto a direção de rotação das rodas afetam a direção geral em que o robô se move. Neste caso, ambos os motores foram designados para ter a mesma potência e direção, mas a forma como os motores transferem o movimento para as rodas faz com que o robô gire em vez de avançar.

> Consulte a seção de Introdução ao Movimento da REV Robotics para obter mais informações sobre a mecânica de transferência de movimento e potência.

Na caixa de informações anterior, foi solicitado que você determinasse em que direção o robô girava. O robô pivota na direção do motor invertido. Por exemplo, quando o motor direito é o motor invertido, o robô irá pivotar para a direita. Se o motor esquerdo for o motor invertido, o robô pivotará para a esquerda.

O efeito dos motores do sistema de transmissão no movimento]



Para a classe Bot, em que o robô pivota para a direita, o motor direito será invertido. Adicione a linha rightmotor.setDirection(DcMotorSimple.Direction.REVERSE); ao op mode, logo abaixo das declarações de variáveis.

```
public void runOpMode() {
        float x;
        double y;

        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");

        rightmotor.setDirection(DcMotorSimple.Direction.REVERSE);

        waitForStart();

        while (opModeIsActive()) {
            rightmotor.setPower(1);
            leftmotor.setPower(1);
          }
    }
```

Adicionar a linha de código rightmotor.setDirection(DcMotorSimple.Direction.REVERSE); inverte a direção do motor direito. Agora, ambos os motores consideram a mesma direção como a direção para a frente.

Estilo Arcade de condução

Lembre-se de que quando os motores estavam girando em direções opostas, o robô girava em círculos. Essa mesma lógica será usada para controlar o robô usando o estilo arcade de controle mencionado na seção Hello Robot - Autonomous Robot.

Programando com controle

Para começar, crie duas variáveis double chamadas drive e turn.

```
public void runOpMode() {
        double x;
        double y;

        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");

        rightmotor.setDirection(DcMotorSimple.Direction.REVERSE);

        waitForStart();
```

Atribua a y o valor -gamepad1.right_stick_y;, que corresponde ao eixo y do joystick direito.

> Lembre-se de que os valores positivos/negativos inseridos pelo eixo y do gamepad são inversos dos valores positivos/negativos do motor.

Atribua x como x = gamepad1.right_stick_x;, que é o eixo x do joystick direito do gamepad. O eixo x do joystick não precisa ser invertido.

```
while (opModeIsActive()) {
        x = gamepad1.right_stick_x;
        y = -gamepad1.right_stick_y;

        rightmotor.setPower(1);
        leftmotor.setPower(1);
        }
```

Definir x = gamepad1.right_stick_x; e y = -gamepad1.right_stick_y; atribui valores do joystick do gamepad a x e y. Como mencionado anteriormente, o joystick fornece valores ao longo de um sistema de coordenadas bidimensional. y recebe o valor do eixo y e x recebe o valor do eixo x. Ambos os eixos geram valores entre -1 e 1.

Para entender melhor, considere a seguinte tabela. A tabela mostra o valor esperado gerado ao mover o joystick completamente em uma direção, ao longo do eixo. Por exemplo, quando o joystick é empurrado completamente na direção para cima, os valores das coordenadas são (0,1).

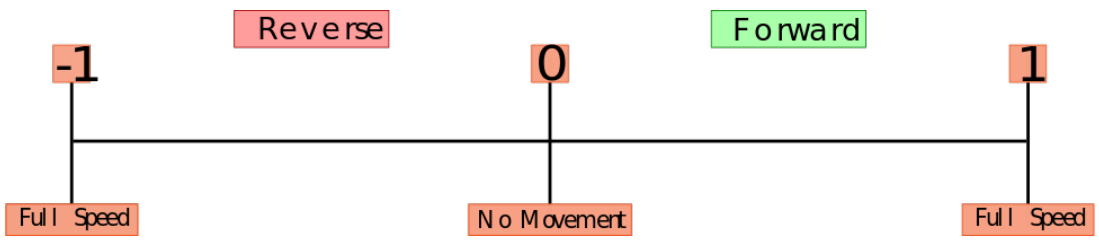
> A tabela abaixo assume que o valor de y foi invertido no código

Joystick Direction	x	y
	0	1
	0	-1
	-1	0
	1	0

Agora que você tem uma compreensão melhor de como o movimento físico do gamepad afeta as entradas numéricas fornecidas ao seu sistema de controle, é hora de considerar como controlar o sistema de propulsão usando o joystick.

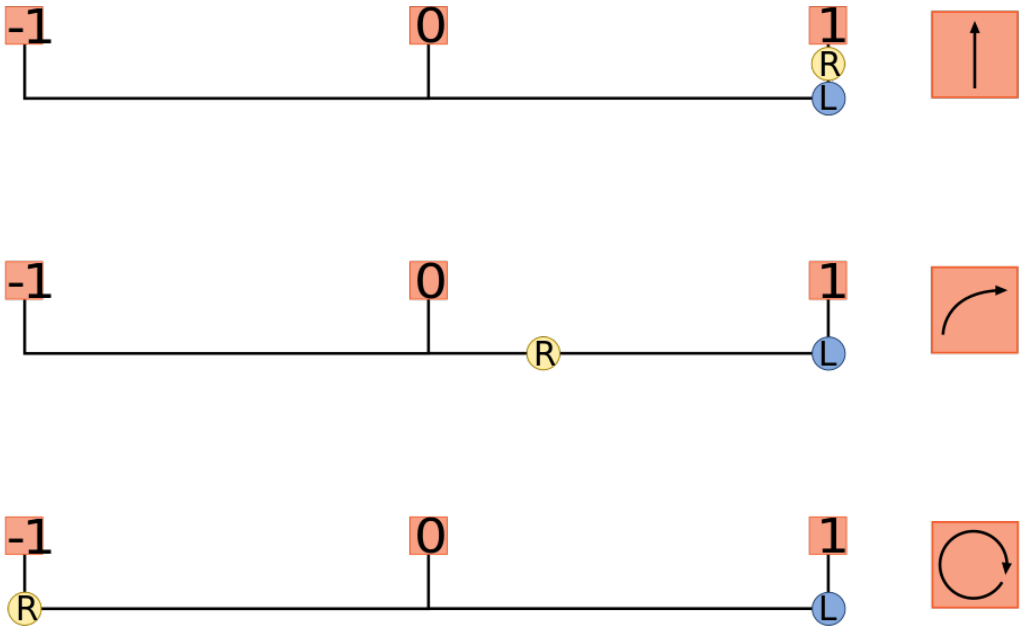
> Lembre-se, da seção de Programação dos Motores da Transmissão, que a velocidade e a direção de um motor desempenham um papel importante na forma como a transmissão se move.

Os resultados numéricos para setPower determinam a velocidade e direção dos motores. Por exemplo, quando ambos os motores são ajustados para 1, eles se movem na direção para frente a toda velocidade (ou 100% do ciclo de trabalho). Assim como nos gamepads, o valor numérico para setPower está em uma faixa de -1 a 1. O valor absoluto do número atribuído determina a porcentagem do ciclo de trabalho. Como exemplo, 0,3 e -0,3 indicam ambos que o motor está operando com um ciclo de trabalho de 30%. O sinal do número indica a direção em que o motor está girando. Para entender melhor, considere o seguinte gráfico.



Quando um motor recebe um valor de setPower entre -1 e 0, o motor girará na direção que considera ser reversa. Quando um motor recebe um valor entre 0 e 1, ele girará para frente.

Na seção de Programação dos Motores da Transmissão, discutiu-se que um robô gira quando os motores estão se movendo em direções opostas. No entanto, isso tem mais a ver com a velocidade e direção. Numericamente, uma transmissão diferencial vai girar para a direita quando o valor de setPower para o motor direito for menor do que o do motor esquerdo. Isso é exemplificado no seguinte exemplo.



Quando ambos os motores estão configurados como rightmotor.setPower(1); leftmotor.setPower(1);, o robô irá se mover a toda velocidade em linha reta. No entanto, quando o rightmotor está se movendo na mesma direção, mas a uma velocidade menor, como rightmotor.setPower(0.3); leftmotor.setPower(1);, o robô irá virar ou girar para a direita. Isso provavelmente resultará em um movimento de arco que não é tão acentuado quanto um pivô completo. Em contraste, quando o rightmotor está configurado para velocidade máxima, mas na direção oposta à do leftmotor, o robô gira para a direita. Portanto, matematicamente, o seguinte é considerado verdadeiro:

Expressão	Resultado
rightmotor.setPower = leftmotor.setPower	Frente ou trás
rightmotor.setPower > leftmotor.setPower	Rotação para esquerda
rightmotor.setPower < leftmotor.setPower	Rotação para direita

Como mencionado anteriormente, gamepad1.right_stick_y e gamepad1.right_stick_x enviam valores para o sistema de controle a partir do joystick do gamepad. Em contraste, a função setPower interpreta informações numéricas definidas no código e envia a corrente apropriada para os motores para ditar o comportamento dos motores.

Em um sistema de direção de arcade, as seguintes entradas (direções) do joystick precisam corresponder às seguintes saídas (valores de potência do motor).

Joystick Direction	(X,Y)	rightmotor	leftmotor
	(0,1)	1	1
	(0,-1)	-1	-1
	(-1,0)	1	-1
	(1,0)	-1	1

Para obter as saídas expressas na tabela acima, os valores do gamepad devem ser atribuídos a cada motor de maneira significativa, onde princípios algébricos podem ser usados para determinar as duas fórmulas necessárias para obter os valores. No entanto, as fórmulas estão fornecidas abaixo.

leftMotor = y - x

rightMotor = y + x

Em vez de setPower(1);, ambos os motores podem ser configurados com as fórmulas mencionadas anteriormente. Por exemplo, o motor direito pode ser configurado como rightmotor.setPower(y - x);

```
while (opModeIsActive()) {
        x = gamepad1.right_stick_x;
        y = -gamepad1.right_stick_y;

        rightmotor.setPower(y-x);
        leftmotor.setPower(y+x);
        }
```

Com isso, você agora possui um controle remoto funcional com sistema de direção arcade. A partir daqui, você pode começar a adicionar o mapeamento de hardware para outras peças do hardware

do robô. Abaixo está um esboço do código esperado para a classe Bot com um mapeamento completo de hardware.

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

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.Blinker;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.Gyroscope;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;
import com.qualcomm.robotcore.util.ElapsedTime;

@TeleOp

public class DualDrive extends LinearOpMode {
    private Blinker control_Hub;
    private DcMotor arm;
    private Servo claw;
    private Gyroscope imu;
    private DcMotor leftmotor;
    private DcMotor rightmotor;
    private DigitalChannel touch;


    @Override
    public void runOpMode() {
        double x;
        double y;

        control_Hub = hardwareMap.get(Blinker.class, "Control Hub");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        touch = hardwareMap.get(DigitalChannel.class, "touch");
```

```
    rightmotor.setDirection(DcMotorSimple.Direction.REVERSE);

    telemetry.addData("Status", "Initialized");
    telemetry.update();

    // Wait for the game to start (driver presses PLAY)
    waitForStart();

    // run until the end of the match (driver presses STOP)
    while (opModeIsActive()) {
        x = gamepad1.right_stick_x;
        y = -gamepad1.right_stick_y;

        rightmotor.setPower(y-x);
        leftmotor.setPower(y+x);

       telemetry.addData("Status", "Running");
       telemetry.update();

    }
}
```

Tempo Decorrido - OnBot Java

Introdução ao tempo decorrido

Uma maneira de criar um código autônomo é usar um temporizador para definir quais ações devem ocorrer quando. Dentro do SDK, as ações podem ser programadas para um temporizador usando o ElapsedTime. Os temporizadores consistem em duas categorias principais: contagem regressiva e contagem progressiva. Na maioria das aplicações, um temporizador é considerado um dispositivo que conta regressivamente a partir de um intervalo de tempo especificado, como o temporizador em um telefone ou micro-ondas. No entanto, alguns temporizadores, como cronômetros, contam progressivamente a partir de zero. Esses tipos de temporizadores medem o tempo decorrido. O ElapsedTime é um temporizador de contagem progressiva, registrando o tempo decorrido desde o início de um evento definido, como o início de um cronômetro. Neste caso, é o tempo decorrido desde quando o temporizador é instanciado ou redefinido no código. O temporizador ElapsedTime começa a contar o tempo decorrido a partir do ponto de sua criação dentro de um código. Por exemplo, nesta seção, o ElapsedTime será criado (ou instanciado) na seção de código que ocorre quando o modo operacional é inicializado. Não há opção de parar o temporizador ElapsedTime. Em vez disso, a função reset() pode ser usada dentro do seu código para reiniciar o temporizador em vários intervalos. Depois que o temporizador é redefinido, o tempo decorrido pode ser consultado chamando métodos como time(), seconds() ou milliseconds(). O tempo fornecido pelos métodos consultados pode ser usado em loops para determinar quanto tempo uma ação específica deve ocorrer.

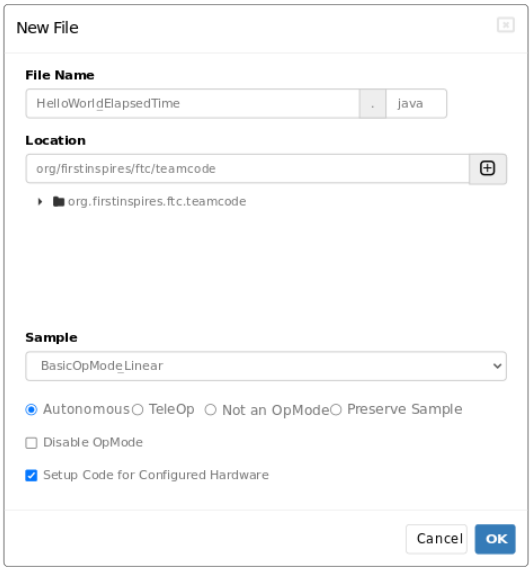
> Para obter mais informações sobre o objeto ElapsedTime, consulte a documentação do Java (Java Docs).

Seções	Objetivos das seções
Noções básicas de programação com tempo	Aprender a lógica para fazer um código autônomo por tempo

Programação com tempo decorrido

Para começar, crie um novo modo operacional (op mode) chamado HelloWorld_ElapsedTime utilizando o exemplo BasicOpMode_Linear. Existem outras características que você pode selecionar para facilitar as coisas ao começar a desenvolver seus modos operacionais autônomos. Por exemplo, como você pode se lembrar, selecionar "Setup Code for Configured Hardware" cria as referências necessárias para o mapa de hardware. Outra opção que você pode escolher é configurar o código como um modo operacional autônomo. Isso adiciona a anotação @Autonomous, que distingue o código como um modo operacional autônomo na aplicação Driver Station.

> Ao criar um modo operacional (op mode), é necessário decidir se ele deve ser configurado como modo autônomo. Para aplicações com duração inferior a 30 segundos, geralmente necessárias para jogabilidade competitiva, é recomendado alterar o tipo de op mode para autônomo. Para aplicações com duração superior a 30 segundos, configurar o código como o tipo de op mode autônomo limitará o tempo de execução do seu código autônomo a 30 segundos. Se você planeja ultrapassar os 30 segundos incorporados no SDK, é recomendável manter o código como um tipo de op mode teleoperado. Para obter informações sobre como os op modes funcionam, visite a seção de Introdução à Programação.



A seleção das características discutidas acima permitirá que você comece com o seguinte código:

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


import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
```

```
import com.qualcomm.robotcore.hardware.AnalogInput;
import com.qualcomm.robotcore.hardware.Gyroscope;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;


@Autonomous

public class HelloWorld_ElapsedTime extends LinearOpMode {
    private DcMotor leftMotor;
    private DcMotor rightMotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftMotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightMotor = hardwareMap.get(DcMotor.class, "rightmotor");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        telemetry.addData("Status", "Initialized");
        telemetry.update();

        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()){
            telemetry.addData("Status", "Running");
```

```
            telemetry.update();
        }
    }
}
```

Como o foco desta seção é o Tempo Decorrido, é necessário criar uma variável de ElapsedTime e uma instância de ElapsedTime. Para fazer isso, a seguinte linha é necessária:

```
private ElapsedTime     runtime = new ElapsedTime();
```

A linha acima realiza duas ações. Uma variável privada ElapsedTime chamada "runtime" é criada. Uma vez que "runtime" é criado e definido como uma variável ElapsedTime, ele pode armazenar as informações e dados de tempo relevantes. A outra parte da linha runtime = new ElapsedTime(); cria uma instância do objeto temporizador ElapsedTime e a atribui à variável "runtime".

Adicione essa linha ao op mode junto com as outras variáveis privadas.

```
public class HelloWorld_ElapsedTime extends LinearOpMode {
    private DcMotor leftMotor;
    private DcMotor rightMotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;
    private ElapsedTime     runtime = new ElapsedTime();
```

O objetivo deste exemplo é realizar uma série de ações em intervalos de tempo, como avançar por três segundos. Outra maneira de pensar é que o robô avança enquanto o temporizador ElapsedTime for menor ou igual a três segundos, ou seja, runtime.seconds() <= 3.0. Para este exemplo em particular, a melhor maneira de atingir esse objetivo é usar um loop while. Substitua o loop while padrão do op mode pelo seguinte loop:

```
        waitForStart();
        while (runtime.seconds() <= 3.0) {


        }
```

> É importante saber que, dentro de um modo operacional linear, um loop while deve sempre ter a condição Booleana opModeIsActive(). Essa condição garante que o loop while será encerrado quando o botão de parada for pressionado.

Os loops while executam quando a condição é verdadeira e param quando a condição é falsa. Neste caso, o loop while deve iniciar apenas se ambas as condições (opModeIsActive() e runtime.seconds() <= 3.0) forem verdadeiras. O loop while deve terminar quando runtime.seconds() > 3 for maior que três segundos ou quando o botão de parada na estação do piloto for pressionado. Para realizar isso, o operador lógico && precisa ser utilizado.

> O operador && é um operador lógico em Java. Este símbolo é equivalente a "e" em Java. Utilizar isso em uma instrução condicional requer que ambas as declarações precisem ser verdadeiras para que a condição geral seja verdadeira.

```
waitForStart();
while (opModeIsActive() && (runtime.seconds() <= 3.0)) {


 }
```

Lembre-se de que o temporizador ElapsedTime começa a contar quando é instanciado ou resetado. Como o temporizador está sendo instanciado quando a variável runtime está sendo criada, e as criações de variáveis estão ocorrendo antes do comando waitForStart(); ser executado; o temporizador começará a contar quando o modo operacional for inicializado, em vez de quando o modo operacional for iniciado. Isso pode causar problemas na consistência do desempenho do robô, dependendo do atraso entre a inicialização e o início.

> Considere o seguinte cenário: Em um ambiente de competição, equipes frequentemente são obrigadas a inicializar seu robô antes do início de uma partida. Isso significa que um robô pode permanecer na fase de inicialização por alguns segundos a alguns minutos. Se um código autônomo é baseado no uso de um temporizador ElapsedTime que começa ao ser instanciado, quanto mais tempo um robô passa na fase de inicialização, menos provável é que ele funcione conforme esperado.

Para evitar problemas decorrentes de um atraso de tempo entre a inicialização e o início, é possível adicionar um reset do temporizador ao código. Adicione a linha runtime.reset(); entre o comando waitForStart(); e o loop while.

```
waitForStart();
runtime.reset();
while (opModeIsActive() && (runtime.seconds() <= 3.0)) {


 }
```

Agora que o temporizador está resetado, vamos em frente e adicionar o código relacionado aos motores. Se você se lembra do artigo "Programming Drivetrain Motors", os motores no trem de força (drivetrain) se espelham entre si. A natureza espelhada da montagem dos motores faz com

que eles girem em direções opostas. Para corrigir essa discrepância, a direção do motor direito precisa ser invertida. Adicione as seguintes linhas de código ao op mode acima do comando waitForStart();

```
rightMotor.setDirection(DcMotor.Direction.REVERSE);
```

Agora, dentro do loop while, adicione as linhas leftmotor.setPower(1); e rightmotor.setPower(1); para definir ambos os motores para funcionar em velocidade máxima na direção para frente.

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

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.AnalogInput;
import com.qualcomm.robotcore.hardware.Gyroscope;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;
import com.qualcomm.robotcore.util.ElapsedTime;



@Autonomous

public class HelloWorld_ElapsedTime extends LinearOpMode {
    private DcMotor leftMotor;
    private DcMotor rightMotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;
    private ElapsedTime     runtime = new ElapsedTime();

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftMotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightMotor = hardwareMap.get(DcMotor.class, "rightmotor");
```

```
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        rightMotor.setDirection(DcMotor.Direction.REVERSE);

        telemetry.addData("Status", "Initialized");
        telemetry.update();
        // Wait for the game to start (driver presses PLAY)

        waitForStart();
        // run until the end of the match (driver presses STOP)

        runtime.reset();
        while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
            leftMotor.setPower(1);
            rightMotor.setPower(1);
        }
```

Agora você tem o código básico necessário para fazer com que seu robô avance por três segundos. Isso deve proporcionar a você uma noção básica de codificação com ElapsedTime. Outras ações, como abrir e fechar uma garra ou levantar um braço, podem ser codificadas em seu programa autônomo.

Como aconselhado nas seções anteriores, é benéfico adicionar telemetria a determinado código para obter os dados de feedback que você deseja ou precisa. Para este exemplo, a telemetria mostrará quantos segundos se passaram para cada etapa da jornada do robô.

```
while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
    leftMotor.setPower(1);
    rightMotor.setPower(1);
    telemetry.addData("Leg 1", runtime.seconds());
    telemetry.update();
        }
```

Para este guia específico, o objetivo final é testar a precisão de um robô avançando do ponto A para o ponto B e, em seguida, recuando de volta para o ponto A. Para fazer isso, é necessário escrever outra seção de código com base no temporizador. Uma maneira de fazer isso é copiar o loop while que você já criou e fazer as edições necessárias, como alternar a direção de energia para os motores.

```
runtime.reset();
while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
    leftMotor.setPower(1);
    rightMotor.setPower(1);
    telemetry.addData("Leg 1", runtime.seconds());
    telemetry.update();
        }

runtime.reset();
while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
    leftMotor.setPower(-1);
    rightMotor.setPower(-1);
    telemetry.addData("Leg 2", runtime.seconds());
    telemetry.update();
        }
```

> Observe que um runtime.reset(); adicional foi adicionado ao código acima. A outra opção para um segundo while loop teria envolvido adicionar uma condição adicional ao while loop, como:
>
> - while(opModeIsActive() && (runtime.seconds() > 3.0) && runtime.seconds() <=6.0)
>
> A escolha de redefinir o temporizador antes de iniciar uma nova etapa da jornada do robô foi feita para reduzir a quantidade de alterações de código que podem ser necessárias durante os testes do código.

Exemplo de código completo]

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

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.hardware.AnalogInput;
import com.qualcomm.robotcore.hardware.Gyroscope;
import com.qualcomm.robotcore.hardware.ColorSensor;
import com.qualcomm.robotcore.hardware.Servo;
import com.qualcomm.robotcore.hardware.DigitalChannel;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
```

```
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;
import com.qualcomm.robotcore.util.ElapsedTime;


@Autonomous

public class HelloWorld_ElapsedTime extends LinearOpMode {
    private DcMotor leftMotor;
    private DcMotor rightMotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;
    private ElapsedTime     runtime = new ElapsedTime();

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftMotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightMotor = hardwareMap.get(DcMotor.class, "rightmotor");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");
        leftMotor.setDirection(DcMotor.Direction.FORWARD); // Set to REVERSE if using AndyMark motors
        rightMotor.setDirection(DcMotor.Direction.REVERSE);

        telemetry.addData("Status", "Initialized");
        telemetry.update();
        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)

        runtime.reset();
        while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
            leftMotor.setPower(1);
            rightMotor.setPower(1);
```

```
            telemetry.addData("Leg 1", runtime.seconds());
            telemetry.update();
        }

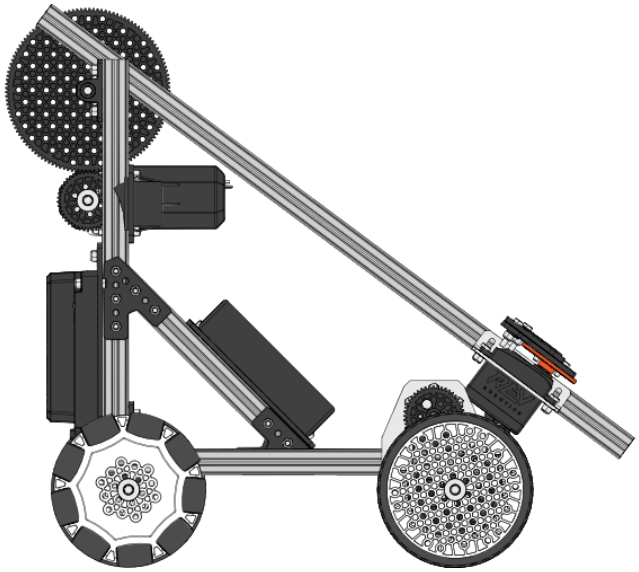
        runtime.reset();
        while (opModeIsActive() && (runtime.seconds() <= 3.0)) {
            leftMotor.setPower(-1);
            rightMotor.setPower(-1);
            telemetry.addData("Leg 2", runtime.seconds());
            telemetry.update();
        }

        }
    }
```

Controle de braço - OnBot Java

Introdução ao controle de braço

O controle de robôs assume muitas formas diferentes. Agora que você passou pela programação de um drivetrain, podemos aplicar esses conceitos ao controle de outros mecanismos. Como este guia utiliza a classe Bot, o foco será nos conceitos básicos de controlar o seu mecanismo principal, um braço de articulação única.



Controlar um braço requer um processo mental diferente daquele que você usou para controlar o drivetrain. Enquanto o drivetrain usa o movimento de rotação dos motores para percorrer uma distância linear, um braço gira em torno de um ponto central, ou junta. Ao trabalhar com um braço, você precisará ter cautela com as limitações físicas do robô, incluindo capacidade de carga, amplitude de movimento e outras forças que possam ser aplicadas.

Nesta seção, você aprenderá a usar os controles do D-pad do gamepad e o Sensor de Toque instalado para controlar o braço. No entanto, o foco desta seção é usar o código para limitar a amplitude de movimento do braço.

Seção	Objetivos da seção
Noções básicas de programação sobre o braço	Introdução à codificação de um braço para controle teleoperado e trabalho com um interruptor de limite.
Programando um braço para uma posição	Utilizando codificadores de motor para mover um braço para uma posição específica, como de 45 graus para 90 graus.
Utilizando limites para controlar a faixa do motor	Trabalhando com os fundamentos do controle de braço, codificador de motor e interruptores de limite para controlar a amplitude de movimento de um braço.

Noções básicas de programação sobre o braço

Comece criando um modo operacional básico chamado HelloRobot_ArmControl.

> Para obter mais informações sobre como criar um modo operacional (op mode), consulte a seção Test Bed - Onbot Java.

Diferentemente do joystick, que envia valores correspondentes à posição do joystick, o Dpad no gamepad envia valores booleanos FALSE/TRUE. Para determinar como o braço se move quando você pressiona DpadUp ou DpadDown, é necessário usar uma instrução if/else if. Crie uma instrução if/else if semelhante à abaixo:

```
while (opModeIsActive()) {
  if(gamepad1.dpad_up){


          }
  else if (gamepad1.dpad_down){


          }
   }
```

Agora que a estrutura básica está no lugar, podemos adicionar os blocos necessários para ditar a direção do braço. A melhor prática é fazer com que o braço se mova para cima quando DpadUp é selecionado e para baixo quando DpadDown é selecionado. Para fazer isso, vamos adicionar arm.setPower(); a cada parte executável da instrução if/else if.

Lembre-se de que o valor atribuído a setPower dita a direção e a velocidade do motor. Entre o motor e a engrenagem no robô classe, um valor positivo moverá o braço para cima, enquanto um valor negativo moverá o braço para baixo. Se você não tem certeza sobre a direção do seu motor, crie o seguinte código e teste para garantir que o motor esteja se comportando conforme o esperado:

```
if(gamepad1.dpad_up){
    arm.setPower(0.2);
      }
else if (gamepad1.dpad_down){
    arm.setPower(-0.2);
      }
```

Iniciar com um ciclo de trabalho mais baixo, como o 0,2 mostrado no código acima, permitirá testes mais fáceis ao tomar decisões para o braço. Mais tarde, neste guia, alteraremos para um ciclo de trabalho mais alto.

> Guarde o modo de operação e tente executar o código. Considere as seguintes perguntas.
>
> - O que acontece se você pressionar para cima no Dpad?
> - O que acontece se você pressionar para baixo no Dpad?

Atualmente, a lógica da instrução if/else if declara que quando gamepad1.dpad_up é verdadeiro (foi pressionado), o motor irá funcionar na direção para frente (ou, neste caso, para cima) a 20% de ciclo de trabalho. Se gamepad1.dpad_down for verdadeiro, o motor irá funcionar reversamente a 20% de ciclo de trabalho. Se você executou o código até este ponto, pode ter notado que mesmo quando você soltou o Dpad, o motor continuou a funcionar na direção selecionada. A instrução if/else if atual diz ao robô quando o motor deve se mover e em que direção, mas nada diz ao motor para parar, assim o braço continua a se mover sem limites.

Para corrigir isso, edite a instrução if/else if para incluir uma ação a ser realizada se nenhuma das condições do gamepad for verdadeira. Como queremos que o braço pare de se mover se nenhuma das condições do gamepad for atendida, vamos usar arm.setPower(0); para parar o motor.

```
if(gamepad1.dpad_up){
    arm.setPower(0.2);
      }
else if (gamepad1.dpad_down){
    arm.setPower(-0.2);
```

```
        }
else {
    arm.setPower(0);
        }
```

> Tente salvar e executar novamente o modo de operação. Preste atenção na velocidade do braço subindo em comparação com descendo. A velocidade parece a mesma?

Trabalhar com um braço introduz diferentes fatores a serem considerados em comparação com o que você viu anteriormente com trens de acionamento. Por exemplo, você notou alguma diferença nas velocidades ao mover o braço para cima ou para baixo? Ao contrário do trem de acionamento, onde o efeito da gravidade impacta os motores de maneira consistente em qualquer direção, a gravidade desempenha um papel significativo na velocidade do motor do braço.

Adicionando um interruptor de limite

Outra consideração a ser feita são as limitações físicas do mecanismo do braço. Certos mecanismos podem ter uma limitação física, e quando essa limitação é ultrapassada, há o risco de danificar o mecanismo ou outro componente do robô. Existem algumas maneiras de limitar o mecanismo com sensores que ajudarão a reduzir a possibilidade de um mecanismo ultrapassar suas limitações físicas. Nesta seção, vamos nos concentrar no uso de um interruptor de limite para restringir a faixa de movimento do braço.

> Esta seção pressupõe que você tenha um conhecimento básico sobre interruptores de limite a partir da seção de Testes e da artigo sobre Sensores Digitais.

Como você pode se lembrar da seção Testes, os interruptores de limite usam lógica booleana para indicar quando um limite foi atingido. Os interruptores de limite geralmente se apresentam na forma de sensores digitais, como o Sensor de Toque, já que os sensores digitais reportam um sinal booleano de ligado/desligado para o sistema, assim como um interruptor de luz.

Se você estiver usando um robô da Classe Bot, seu robô deve ter um Sensor de Toque montado na frente do chassi. Você também deve ter instalado um Limit Switch Bumper. Juntos, esses itens formam um sistema de interruptor de limite. Ao utilizar o sistema de interruptor de limite, você pode evitar que o braço do seu robô Classe Bot ultrapasse o limite físico inferior, ou o que será conhecido como nossa posição inicial. Vamos começar a programar!

> Antes de prosseguir com o código, certifique-se de que o mecanismo está interagindo com e pressionando o Sensor de Toque. Se você estiver usando a Classe Bot, isso envolve garantir que o chassi esteja pressionando ativamente o Sensor de Toque quando o braço descer.

No trecho "Banco de testes - Onbot Java", você aprendeu como criar um programa básico de interruptor de limite, semelhante ao exemplo abaixo.

```
 if (touch.getState()){
    //Touch Sensor is not pressed
     arm.setPower(0.2);


} else {
    //Touch Sensor is pressed
    arm.setPower(0);
                    }
```

Se você se recorda da seção inicial sobre interruptores de limite, o Sensor de Toque opera em um estado binário de FALSO/VERDADEIRO. Quando o sensor de toque não está pressionado, touch.getState() retorna verdadeiro; quando o sensor de toque está pressionado, touch.getState() retorna falso. A lógica do código afirma que quando o sensor de toque não está pressionado, o motor funciona com um ciclo de trabalho de 20%.

Em vez de fazer o motor funcionar com um ciclo de trabalho de 20% quando o Sensor de Toque não está pressionado e parar quando o sensor é pressionado, queremos controlar o braço usando o gamepad ainda. Para fazer isso, podemos aninhar a instrução if/else if do Gamepad dentro da instrução if/else do Interruptor de Limite.

> Para esta próxima parte, estaremos utilizando a instrução if/else if criada nas seções Basics of Programming and Arm. Daqui para frente, essa lógica básica de código será referida como a instrução if/else if do Gamepad. O código do interruptor de limite será conhecido como a instrução if/else do Interruptor de Limite. Ambas as partes do código serão referenciadas novamente.

```
if(gamepad1.dpad_up){
      arm.setPower(0.2);
         }
else if (gamepad1.dpad_down){
      arm.setPower(-0.2);
         }
else {
      arm.setPower(0);
         }
```

Segundo código:

```
if(touch.getState()){
    if(gamepad1.dpad_up){
        arm.setPower(0.2);
            }
    else if (gamepad1.dpad_down){
        arm.setPower(-0.2);
            }
    else {
        arm.setPower(0);
            }
     }
  else {
    arm.setPower(0);
     }
```

> Salve o modo operacional e execute-o. O que acontece quando o Sensor de Toque é pressionado?

Uma das características comuns de um interruptor de limite, como o sensor de toque, é a capacidade de redefinir para o seu estado padrão. Se você pressionar o sensor de toque com o dedo, pode notar que assim que liberar a pressão que está aplicando, o sensor de toque voltará ao seu estado padrão de "não pressionado". No entanto, é necessário liberar a pressão para conseguir isso.

> Certifique-se de que o mecanismo está efetivamente interagindo com o Sensor de Toque. Para o Bot da Classe, talvez seja necessário ajustar o Sensor de Toque para que o Limit Switch Bumper esteja interagindo com ele de forma mais consistente.

O código no bloco de informações acima determina que quando o Sensor de Toque é pressionado, o motor do braço é definido como zero. Isso funcionaria em um mecanismo onde o Sensor de Toque pode retornar ao seu estado padrão por conta própria. No entanto, uma vez que o braço pressiona o Sensor de Toque, o peso do mecanismo impedirá que o Sensor de Toque retorne ao seu estado padrão. A combinação do peso do mecanismo e a lógica do código no bloco de informações significa que, uma vez que o braço atinge seu limite, ele não será capaz de se mover novamente.

Para remediar isso, uma ação para mover o braço na direção oposta ao limite precisa ser adicionada à instrução "else". Como o Sensor de Toque é um limite inferior para o braço, o braço precisará se mover para cima (ou o motor na direção para a frente) para se afastar do sensor de toque. Para fazer isso, podemos criar uma instrução if/else semelhante à nossa instrução if/else do gamepad. Em vez de ter as operações normais do gamepad, quando o Sensor de Toque e o DpadUp são pressionados, o braço se move para longe do Sensor de Toque. Uma vez que o Sensor

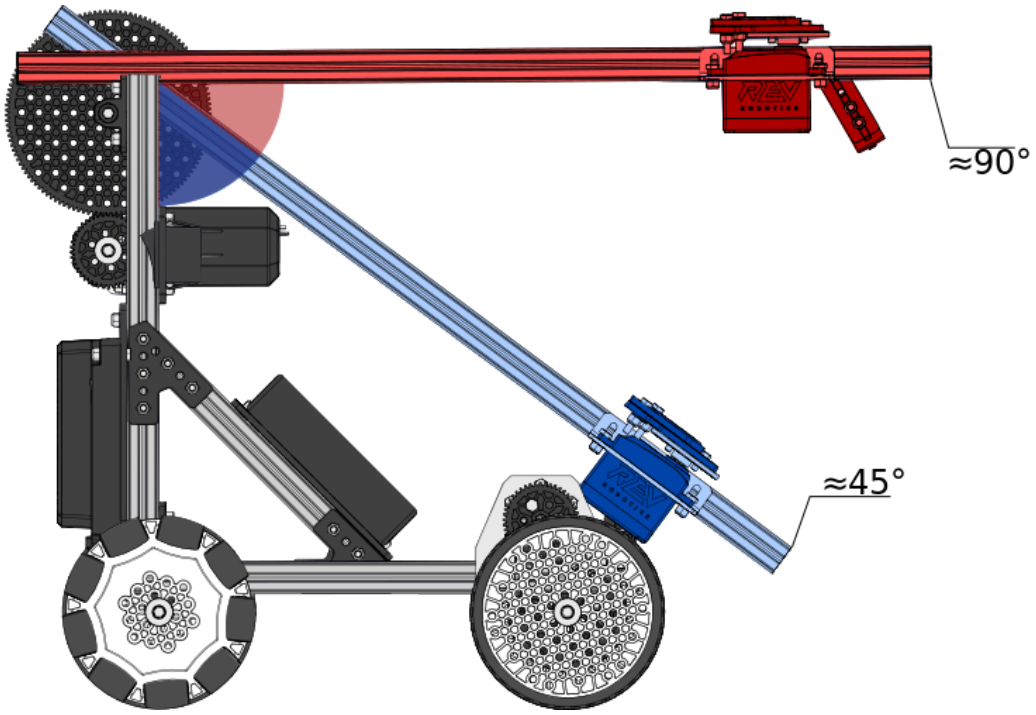
de Toque não relata mais falso, as operações normais do gamepad retornam e o braço pode se mover em qualquer direção novamente.

```
if(touch.getState()){
      if(gamepad1.dpad_up){
           arm.setPower(0.2);
               }
      else if (gamepad1.dpad_down){
           arm.setPower(-0.2);
               }
      else {
           arm.setPower(0);
               }
       }
  else {
       if(gamepad1.dpad_up){
           arm.setPower(0.2);
               }
       else{
           arm.setPower(0);
       }
```

Programando um braço para uma posição

Na seção de Navegação por Encoder, foi introduzido o conceito de mover o motor para uma posição específica com base em ticks do encoder. O processo destacado na Navegação por Encoder focou em como converter de ticks do encoder para rotações e, consequentemente, para uma distância linear. Um procedimento semelhante pode ser utilizado para mover o braço para uma posição específica. No entanto, ao contrário do drivetrain, o braço não segue um caminho linear. Em vez de converter para uma distância linear, faz mais sentido converter os ticks do encoder em um ângulo medido em graus.

Na imagem abaixo, são apresentadas duas posições potenciais para o braço do ClassBot. Uma das posições - destacada em azul abaixo - é a posição em que o braço encontra o limite do sensor de toque. Devido ao limite, esta posição será nossa posição padrão ou de início. No guia de construção do ClassBot, sabe-se que a extrusão que suporta a bateria está em um ângulo de 45 graus. Como o braço está aproximadamente paralelo a essas extrusões quando está na posição inicial, podemos estimar que o ângulo padrão do braço é aproximadamente 45 graus.



O objetivo desta seção é determinar a quantidade de ticks do encoder necessários para mover o braço de sua posição inicial para uma posição em torno de 90 graus. Existem algumas maneiras diferentes de realizar isso. Uma estimativa pode ser feita movendo o braço para a posição desejada e registrando o retorno de telemetria da Estação do Motorista. Outra opção é realizar os cálculos matemáticos para encontrar a quantidade de ticks do encoder que ocorrem por grau movido. Siga esta seção para percorrer ambas as opções e determinar qual é a melhor para a sua equipe.

Estimando a posição do braço

Para estimar a posição do braço utilizando telemetria e testando, vamos começar com uma programação if/else com controle

```
if(gamepad1.dpad_up){
      arm.setPower(0.2);
         }
else if (gamepad1.dpad_down){
      arm.setPower(-0.2);
         }
else {
      arm.setPower(0);
         }
```

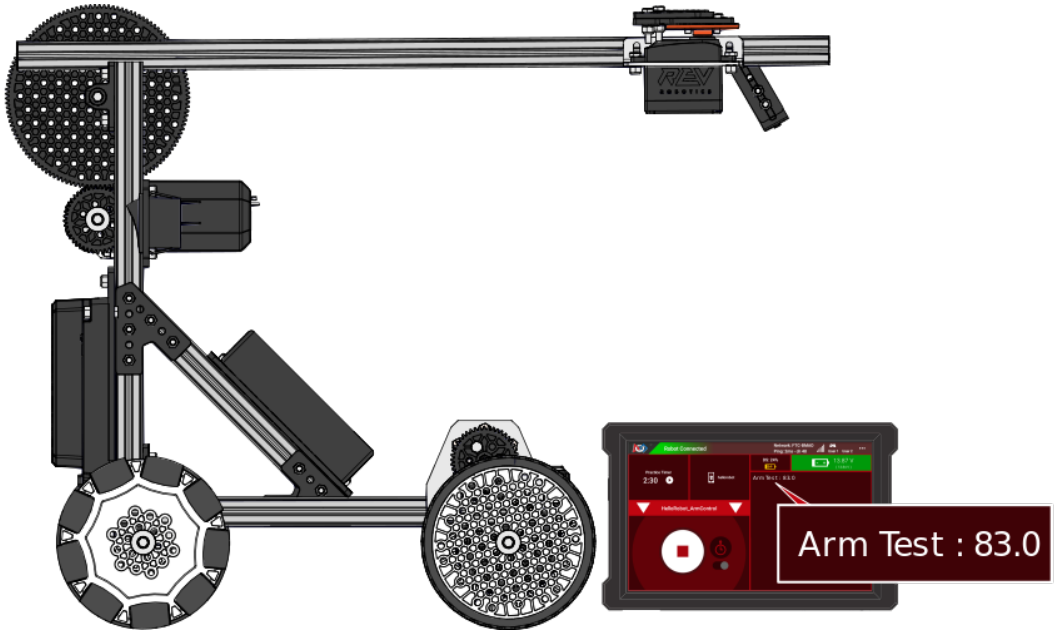
> Por enquanto, você pode comentar o código relacionado ao interruptor de limite.

Dentro do loop while, adicione a linha telemetry.addData("Arm Test", arm.getCurrentPosition()); e telemetry.update();.

```
while(opModeIsActive){
    if(gamepad1.dpad_up){
      arm.setPower(0.2);
          }
     else if (gamepad1.dpad_down){
      arm.setPower(-0.2);
          }
     else {
      arm.setPower(0);
          }
     telemetry.addData("Arm Test", arm.getCurrentPosition());
     telemetry.update();

 }
```

Guarde o modo operacional (op mode) e execute-o. Utilize os comandos do gamepad para mover o braço para a posição de 90 graus. Assim que o braço estiver devidamente posicionado, leia as informações de telemetria na Driver Station para determinar a contagem do codificador em relação à posição do braço.



> Recordem-se da seção Conceitos Básicos de Encoder, na qual a posição do encoder é definida como 0 cada vez que o Control Hub é ligado. Isso significa que, se o braço estiver em uma posição diferente da posição inicial quando o

> Control Hub for ligado, essa posição se tornará zero em vez da posição inicial. O número fornecido na imagem acima não é necessariamente uma contagem precisa do encoder para a posição de 90 graus. Para obter a leitura mais precisa do encoder para o seu robô, certifique-se de que a posição inicial seja registrada como 0 contagens do encoder. Para aumentar ainda mais a precisão, considere realizar várias execuções de teste antes de decidir sobre o número de contagens.

Recordem-se de que, para executar RUN_TO_POSITION, as seguintes três linhas de código precisam ser adicionadas a ambas as seções do bloco if/else if do Gamepad.

```
arm.setTargetPosition(0);
arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
arm.setPower(0);
```

Quando DpadUp for pressionado, o braço deve se mover para a posição de 90 graus. Quando DpadDown for pressionado, o braço deve voltar à posição inicial. Para fazer isso, defina a primeira linha arm.setTargetPosition(0); igual ao número de ticks necessários para o seu braço atingir 90 graus. Para este exemplo, vamos usar 83 ticks.

Como queremos que DpadDown retorne o braço à posição inicial, manter arm.setTargetPosition(0); definido como 0 nos permitirá realizar isso. Defina ambas as linhas arm.setPower(0); como 0,5.

```
if(gamepad1.dpad_up){
    arm.setTargetPosition(83);
    arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
    arm.setPower(0.5);
        }
else if (gamepad1.dpad_down){
     arm.setTargetPosition(0);
     arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
     arm.setPower(0.5);
        }
```

> Observação: o código acima tem uma nome de arquivo "Target Position if/else if", esse código vai ser referênciado de novo

> Recordem-se de que a posição alvo (target position) dita em que direção o motor se move, assumindo o controle da direcionalidade anteriormente controlada por arm.setPower(); Portanto, ambos os blocos podem ser definidos com um valor positivo, já que eles

> controlarão a velocidade.

Se você tentar executar este código, pode perceber que o braço oscila na posição de 90 graus. Quando esse comportamento está presente, você também deve notar a saída de telemetria para as contagens do encoder flutuando. RUN_TO_POSITION é um Controle em Malha Fechada, o que significa que, se o braço não atingir perfeitamente a posição alvo, o motor continuará a oscilar até conseguir. Quando os motores continuam a oscilar e nunca atingem completamente a posição alvo, isso pode ser um sinal de que os fatores que determinam as tolerâncias e outros aspectos do loop fechado não estão ajustados para esse motor ou mecanismo específico. Existem maneiras de ajustar o motor, mas por enquanto queremos nos concentrar em trabalhar com o braço e expandir sobre como limites e posições funcionam em relação ao mecanismo.

Calculando a posição alvo

Na introdução inicial ao "run to position", você trabalhou nos cálculos necessários para converter os ticks por rotação de um motor em ticks por milímetro movido. Agora, queremos nos concentrar em como converter os ticks por rotação do motor em ticks por grau movido. A partir da seção anterior, você deve ter uma estimativa aproximada da quantidade de ticks necessários para chegar à posição de 90 graus. O objetivo desta seção é trabalhar na obtenção de uma posição mais exata.

Para começar, você precisará de algumas das mesmas variáveis que usamos em Navegação com Encoder.

Ticks por revolução

Recordem-se de que os ticks por revolução do eixo do encoder são diferentes dos ticks por revolução do eixo que controla um mecanismo. Vimos isso na seção de Navegação com Encoder, quando os ticks por revolução no motor eram diferentes dos ticks por revolução da roda. À medida que o movimento é transmitido de um motor para um mecanismo, a resolução dos ticks do encoder muda.

> Para mais informação sobre o efeito da transmissão em um mecanismo, consulte a seguinte seção:

Redução

A quantidade de ticks por revolução do eixo do encoder depende do motor e do encoder. Os fabricantes de motores com encoders embutidos terão informações sobre a quantidade de ticks por revolução.

> Visite o site da fabricante dos seus motores e encoders para saber as contagens

Nas especificações do Core Hex tem dois tipos diferentes de Contagens por revolução do encoder:

- No motor - **4 contagens/revolução**
- Na saída - **288 contagens/revolução**

No motor, temos o número de contagens do encoder no eixo em que o encoder está instalado. Esse número é equivalente aos 28 counts por revolução que usamos para o HD Hex Motor. Os 288 counts "na saída" levam em consideração a mudança na resolução após o movimento ser transmitido do motor para a caixa de engrenagens embutida de 72:1. Vamos usar o valor 288 como ticks por revolução para que não seja necessário considerar a caixa de engrenagens em nossa variável total de redução de engrenagens.

Redução total

Como incorporamos a redução de engrenagens da caixa de engrenagens do motor nos ticks por revolução, o foco principal desta seção é calcular a redução de engrenagens da junta do braço. O eixo do motor aciona uma engrenagem de 45 dentes que transmite movimento para uma engrenagem de 125 dentes. A relação total de engrenagens é 125D:45D. Para calcular a redução de engrenagens para este conjunto de engrenagens, podemos simplesmente dividir 125 por 45.

$$125/45 = 2.777778$$

Para recapitular, para o Robô V2 as seguinte informações são verídicas:

Descrição	Valor
Ticks por revolução	288 ticks
Redução total	2.777778

Vamos criar duas variáveis agora que temos essa informação:

- COUNTS_PER_MOTOR_REV
- GEAR_REDUCTION

> A convenção comum de nomeação para variáveis constantes é conhecida como CONSTANT_CASE, em que o nome da variável está todo em maiúsculas e as palavras são separadas por um sublinhado.

Adicione as variáveis COUNTS_PER_MOTOR_REV e GEAR_REDUCTION ao op mode abaixo de onde as variáveis de hardware são criadas.

```
public class HelloRobot_ArmControl extends LinearOpMode {
    private DcMotor arm;
```

```
    static final double     COUNTS_PER_MOTOR_REV    = 288;
    static final double     GEAR_REDUCTION    = 2.7778;
```

Agora que essas duas variáveis foram definidas, podemos usá-las para calcular outras duas variáveis: a quantidade de contagens do encoder por rotação da engrenagem acionada de 125 dentes e o número de contagens por grau movido.

Calcular as contagens por revolução da engrenagem de 125 dentes (ou COUNTS_PER_GEAR_REV) é a mesma fórmula usada na Navegação com Encoder para nossa variável COUNTS_PER_WHEEL_REV. Portanto, para obter essa variável, podemos multiplicar COUNTS_PER_MOTOR_REV por GEAR_REDUCTION.

```
static final double  COUNTS_PER_GEAR_REV    = COUNTS_PER_MOTOR_REV * GEAR_REDUCTION;
```

Para calcular o número de contagens por grau movido (ou COUNTS_PER_DEGREE), divida a variável COUNTS_PER_GEAR_REV por 360.

```
static final double     COUNTS_PER_DEGREE    = COUNTS_PER_GEAR_REV/360;
```

Adicione essas variáveis ao Op Mode.

```
public class HelloRobot_ArmControl extends LinearOpMode {
    private DcMotor arm;

    static final double     COUNTS_PER_MOTOR_REV    = 288;
    static final double     GEAR_REDUCTION    = 2.7778;
    static final double     COUNTS_PER_GEAR_REV    = COUNTS_PER_MOTOR_REV * GEAR_REDUCTION;
    static final double     COUNTS_PER_DEGREE    = COUNTS_PER_GEAR_REV/360;
```

Finalmente, precisamos criar uma variável não constante que atuará como nossa posição. Crie uma variável chamada armPosition acima do comando waitForStart();.

```
public void runOpMode() {
        arm = hardwareMap.get(DcMotor.class, "arm");

        int armPosition;

        waitForStart();
```

Adicione essa variável à seção if(gamepad1.dpad_up) da instrução if/else if, já que essa seção indica a posição de 90 graus. Para chegar à posição de 90 graus, o braço precisa se mover aproximadamente 45 graus. Defina a posição do braço como COUNTS_PER_DEGREE vezes 45.

> Recordem-se de que setTargetPosition() requer um número inteiro como seu parâmetro. Ao definir armPosition, lembre-se de adicionar a linha (int) na frente da variável double. No entanto, você precisa ter cautela em relação a possíveis erros de arredondamento. Como COUNTS_PER_MM faz parte de uma equação, é recomendável converter para um número inteiro após encontrar o resultado da equação.
>
> ```
> armPosition = (int)(COUNTS_PER_DEGREE * 45);
> ```

```
while (opModeIsActive()) {


  if(gamepad1.dpad_up){
      armPosition = (int)(COUNTS_PER_DEGREE * 45);
      arm.setTargetPosition(83);
      arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
      arm.setPower(0.4);
           }
```

Defina a posição alvo para *armPosition*:

```
if(gamepad1.dpad_up){
    armPosition = (int)(COUNTS_PER_DEGREE * 45);
    arm.setTargetPosition(armPosition);
    arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
    arm.setPower(0.4);
         }
else if (gamepad1.dpad_down){
     arm.setTargetPosition(0);
     arm.setMode(DcMotor.RunMode.RUN_TO_POSITION);
     arm.setPower(0.4);
         }
```

> Poderíamos alterar para o que armPosition é igual na parte do gamepad1.dpad_down da instrução if/else if, como por exemplo:
>
> ```
> else if (gamepad1.dpad_down){
>   armPosition = (int)(COUNTS_PER_DEGREE * 0);
> ```

```
arm.setTargetPosition(armPosition);
arm.setTargetPosition(armPosition);
arm.setPower(0.4);
}
```

Neste caso, estaríamos constantemente redefinindo armPosition para atender às necessidades das posições que desejamos criar. Uma vez que, no momento, temos apenas duas posições - a posição inicial e a posição de 90 graus - não é necessário. No entanto, é uma boa prática criar uma variável em situações como essa. Se quisermos adicionar outra posição posteriormente, podemos editar facilmente a variável para atender às nossas necessidades.

Utilizando limites para controlar a faixa de um movimento

Nas seções anteriores, você trabalhou em alguns dos fundamentos para restringir o alcance de movimento de um braço. A partir dessas seções, você deveria ter a base necessária para realizar um controle básico do braço. No entanto, existem outras maneiras criativas de usar posições de encoder e limites para expandir o controle sobre o braço.

Esta seção abordará dois tipos adicionais de controle. O primeiro tipo de controle que exploraremos é a ideia de "soft limits" (limites suaves). Na seção de Adição de um Interruptor de Limite, discutimos o conceito de limites físicos de um mecanismo; no entanto, pode haver momentos em que você precisa limitar o alcance de movimento de um braço sem instalar um limite físico. Para fazer isso, pode-se usar código baseado em posição para criar um intervalo para o braço.

Depois de ter uma ideia básica de como criar limites suaves, exploraremos como usar um interruptor de limite (como um sensor de toque) para redefinir o alcance de movimento. Esse tipo de controle reduz o risco de ficar preso fora do intervalo pretendido, o que pode afetar o comportamento esperado do seu robô.

Para definir os limites suaves, usaremos alguma lógica básica que estabelecemos em seções anteriores, com algumas alterações editadas. Comece com um Op Mode básico e adicione as variáveis constantes da seção anterior (calculando a posição alvo).

```
@TeleOp

public class Basic extends LinearOpMode {
    private DcMotor  arm;

    static final double     COUNTS_PER_MOTOR_REV    = 288;
    static final double     GEAR_REDUCTION    = 2.7778;
    static final double     COUNTS_PER_GEAR_REV    = COUNTS_PER_MOTOR_REV * GEAR_REDUCTION;
    static final double     COUNTS_PER_DEGREE    = COUNTS_PER_GEAR_REV/360;

    @Override
    public void runOpMode() {
        arm = hardwareMap.get(DcMotor.class, "arm");



        waitForStart();

        while (opModeIsActive()) {
            telemetry.addData("Status", "Running");
            telemetry.update();

        }
    }
}
```

Depois, nós precisamos criar nossos limites superior e inferior. Portanto, crie duas novas variáveis (int), uma chamada de **minPosition** e a outra de **maxPosition**. Adicione ambos na parte de inicialização do Op Mode, acima do waitForStart();.

```
public void runOpMode() {
        arm = hardwareMap.get(DcMotor.class, "arm");


        int minPostion;
        int maxPosition;
        waitForStart();
```

Agora nós queremos que a **minPosition** defina a posição inicial, e que a **maxPosition** defina a posição de 90°. Defina a **minPosition** igual a 0 e defina **maxPosition** igual a **COUNTS_PER_DEGREE** vezes 45.

> Lembre-se de que é necessário converter os tipos primitivos

```
int minPostion = 0;
int maxPosition = (int)(COUNTS_PER_DEGREE *45);
```

Uma lógica de if/else if precisa ser adicionada no controle do braço, para isso nós podemos usar a mesma lógica que utilizamos em noções básicas da programação de um braço.

```
while(opModeIsActive()){
    if(gamepad1.dpad_up){
        arm.setPower(0.5);
        }
    else if (gamepad1.dpad_down){
        arm.setPower(-0.5);
        }
    else {
        arm.setPower(0);
        }
    }
```

Para definir o limite, precisamos editar nossa instrução if/else if para que os limites sejam incorporados. Se DpadUp for selecionado e a posição do braço for menor que a maxPosition, então o braço se moverá para a maxPosition. Se DpadDown for selecionado e a posição do braço for maior que a minPosition, então o braço se moverá em direção à minPosition.

```
while (opModeIsActive()) {
    if (gamepad1.dpad_up && arm.getCurrentPosition() < maxPosition) {
        arm.setPower(0.5);
        }
    else if (gamepad1.dpad_down && arm.getCurrentPosition() > minPosition) {
        arm.setPower(-0.5);
        }
    else {
        arm.setPower(0);
        }
    }
```

> A configuração atual do código interromperá o motor em qualquer ponto em que as condições para alimentar o motor não sejam atendidas. Dependendo de fatores como o peso do mecanismo e qualquer carga que ele esteja suportando, quando o motor para, o braço

> pode cair abaixo da maxPosition. Reserve um tempo para testar o código e confirmar se ele se comporta da maneira que você espera.

Substítuindo limites

Um dos benefícios de ter um limite suave é a capacidade de ultrapassar esse limite. Como a posição zero dos ticks do encoder é determinada pela posição do braço quando o Control Hub é ligado; se não prestarmos atenção à posição do braço no momento da inicialização, o alcance de movimento do braço é afetado. Por exemplo, se precisarmos reiniciar o Control Hub enquanto o braço estiver na posição de 90 graus, a posição de 90 graus será igual a 0 ticks do encoder. Uma maneira de contornar isso é criar uma substituição para o alcance de movimento.

Existem várias maneiras diferentes de criar uma substituição desse tipo; no nosso caso, vamos usar um botão e um sensor de toque para ajudar a redefinir nosso intervalo.

Comece editando a instrução if/else if para adicionar outra condição else if. Use a linha gamepad1.a como condição. Adicione a linha arm.setPower(-0.5); como a ação correspondente.

```
while (opModeIsActive()) {
    if (gamepad1.dpad_up && arm.getCurrentPosition() < maxPosition) {
            arm.setPower(0.5);
            }
    else if (gamepad1.dpad_down && arm.getCurrentPosition() > minPosition) {
            arm.setPower(-0.5);
            }
    else if(gamepad1.a){
            arm.setPower(-0.5);
    else {
            arm.setPower(0);
            }
    }
```

Agora que temos essa alteração em vigor, quando o botão A é pressionado, o braço se moverá em direção à posição inicial. Quando o braço alcançar e pressionar o sensor de toque, queremos utilizar STOP_AND_RESET_ENCODER.

Podemos criar outra instrução if que se concentra em realizar essa parada e reinicialização quando o sensor de toque é pressionado. Como o sensor de toque relata true quando não está pressionado e false quando está, precisaremos usar o operador lógico not !.

> O operador ! pode ser usado em instruções binárias condicionais quando você precisa inverter se algo é verdadeiro ou falso. Por exemplo, uma instrução if é ativada quando algo é verdadeiro, mas quando touch.getState(); relata verdadeiro, significa que não está

pressionado. No nosso caso, queremos que esta instrução if seja ativada quando o sensor de toque está pressionado, portanto, precisamos usar o operador not.

```
if (!touch.getState()) {
        arm.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
       }
```

Portanto, se o sensor de toque retornar falso (ou estiver pressionado), o modo de execução do motor STOP_AND_RESET_ENCODER será ativado, fazendo com que o encoder do motor seja redefinido para 0 ticks.

Agora que este código está concluído, experimente testá-lo!

```
@TeleOp


public class HelloRobot_ArmControl extends LinearOpMode {
    private DcMotor arm;
    private Servo claw;
    private Gyroscope imu;
    private DcMotor leftmotor;
    private DcMotor rightmotor;
    private DigitalChannel touch;

    static final double     COUNTS_PER_MOTOR_REV    = 288;
    static final double     GEAR_REDUCTION    = 2.7778;
    static final double     COUNTS_PER_GEAR_REV    = COUNTS_PER_MOTOR_REV * GEAR_REDUCTION;
    static final double     COUNTS_PER_DEGREE    = COUNTS_PER_GEAR_REV/360;




    @Override
    public void runOpMode() {
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        int minPostion = 0;
        int maxPosition = (int)(COUNTS_PER_DEGREE *45);
```

```
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()) {
            if (gamepad1.dpad_up && arm.getCurrentPosition() < maxPosition) {
                arm.setPower(0.5);
                    }
            else if (gamepad1.dpad_down && arm.getCurrentPosition() > minPosition) {
                arm.setPower(-0.5);
                    }
            else if (gamepad1.a) {
                arm.setPower(-0.5);
                    }
            else {
                arm.setPower(0);
                }
            if (!touch.getState()) {
              arm.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
                    }
        telemetry.addData("Arm Test", arm.getCurrentPosition());
        telemetry.update();
        }
    }
}
```

Navegação por Encoder

Na seção anterior, você aprendeu sobre como usar o Elapsed Time para permitir que seu robô navegue autonomamente pelo ambiente ao seu redor. Ao começar, muitas das ações do robô podem ser realizadas ligando um motor por um tempo específico. Eventualmente, essas ações baseadas em tempo podem não ser precisas ou repetíveis o suficiente. Fatores ambientais, como o estado da carga da bateria durante a operação e o desgaste dos mecanismos pelo uso, podem afetar todas as ações baseadas em tempo. Felizmente, há uma maneira de fornecer feedback ao robô sobre como ele está operando, usando sensores; dispositivos usados para coletar informações sobre o robô e o ambiente ao seu redor.

Com o Elapsed Time, para mover o robô para uma distância específica, você teve que estimar a quantidade de tempo e a porcentagem de ciclo de trabalho necessária para ir do ponto a ao ponto b. No entanto, os motores REV vêm com encoders embutidos, que fornecem feedback na forma de ticks (ou contagens) por revolução do motor. As informações fornecidas pelos encoders podem ser usadas para mover o motor para uma posição-alvo ou uma distância-alvo.

Mover os motores para uma posição específica, usando os encoders, elimina possíveis imprecisões ou inconsistências do uso do Elapsed Time. O foco desta seção é mover o robô para uma posição-alvo usando encoders.

> Existem dois artigos que abordam os conceitos básicos dos encoders. "Using Encoders" explora os fundamentos dos diferentes tipos de modos de motor, bem como alguns exemplos de aplicação desses modos no código. Nesta seção, concentraremos no uso de RUN_TO_POSITION.

O outro artigo, "Encoders", concentra-se na funcionalidade geral de um encoder.

Recomenda-se que você revise ambos os artigos antes de prosseguir com este guia. Links abaixo.

Utilizando encoders

Encoders

Fundamentos da programação com Encoders

Comece criando um Op Mode simples chamado **HelloRobot_EncoderAuton**

> Ao criar um Op Mode, uma decisão precisa ser tomada quanto a defini-lo ou não como modo autônomo. Para aplicações com duração inferior a 30 segundos, geralmente exigidas para a jogabilidade competitiva, recomenda-se alterar o tipo de Op Mode para autônomo. Para aplicações com duração superior a 30 segundos, definir o código como o tipo de Op Mode autônomo limitará seu código autônomo a 30 segundos de tempo de execução. Se você planeja exceder os 30 segundos incorporados ao SDK, é recomendável manter o código como um tipo de Op Mode teleoperado. Para obter informações sobre como os Op Modes funcionam, visite a seção de Introdução à Programação. Para obter mais informações sobre como alterar o tipo de Op Mode, confira a seção Banco de Testes - OnBot Java.

> A estrutura do Op Mode abaixo é simplificada e inclui apenas os componentes necessários para criar o código baseado em encoders.

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

import com.qualcomm.robotcore.eventloop.opmode.LinearOpMode;
import com.qualcomm.robotcore.eventloop.opmode.Autonomous;
import com.qualcomm.robotcore.eventloop.opmode.TeleOp;
import com.qualcomm.robotcore.eventloop.opmode.Disabled;
import com.qualcomm.robotcore.hardware.DcMotor;
import com.qualcomm.robotcore.hardware.DcMotorSimple;

@Autonomous //sets the op mode as an autonomous op mode

public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;

    @Override
    public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
```

```
        // Wait for the game to start (driver presses PLAY)
        waitForStart();

        // run until the end of the match (driver presses STOP)
        while (opModeIsActive()){

        }
    }
}
```

Assim como em toda navegação relacionada ao drivetrain, a direção de um dos motores precisa ser invertida para que ambos os motores se movam na mesma direção. Como a Classe Bot V2 ainda está sendo usada, adicione a linha rightmotor.setDirection(DcMotor.Direction.REVERSE); ao código abaixo da linha de código rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        waitForStart();
```

> Para obter mais informações sobre a direcionalidade dos motores, confira a página inicial desse capítulo.

Lembrando do uso de encoders que usar o modo RUN_TO_POSITION requer um processo de três etapas. O primeiro passo é definir a posição alvo. Para definir a posição alvo, adicione as linhas leftmotor.setTargetPosition(1000); e rightmotor.setTargetPosition(1000); ao Op Mode após o comando waitForStart();. Para obter uma posição alvo que corresponda a uma distância alvo, são necessários alguns cálculos, que serão abordados posteriormente. Por enquanto, defina a posição alvo como 1000 ticks.

```
waitForStart();

leftmotor.setTargetPosition(1000);
rightmotor.setTargetPosition(1000);

while (opModeIsActive()){
```

```
    }
```

O próximo passo é definir ambos os motores para o modo RUN_TO_POSITION. Adicione as linhas leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION); e rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION); ao seu código, abaixo das linhas de código setTargetPosition.

```
waitForStart();

leftmotor.setTargetPosition(1000);
rightmotor.setTargetPosition(1000);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

while (opModeIsActive()){

    }
```

O foco principal do processo de três etapas é definir uma meta, dizer ao robô para se mover para essa meta e a que velocidade (ou velocidade) o robô deve atingir essa meta. Normalmente, o próximo passo recomendado é calcular a velocidade e definir uma velocidade alvo com base nos ticks. No entanto, isso requer bastante matemática para encontrar a velocidade apropriada. Para fins de teste, é mais importante garantir que a parte principal do código esteja funcionando antes de se aprofundar demais na criação do código. Uma vez que a função setPower foi abordada em seções anteriores e comunicará ao sistema qual velocidade relativa (ou, neste caso, ciclo de trabalho) é necessária para atingir a meta, ela pode ser usada no lugar de setVelocity por enquanto.

Adicione as linhas para definir a potência de ambos os motores para 80% do ciclo de trabalho.

```
waitForStart();

leftmotor.setTargetPosition(1000);
rightmotor.setTargetPosition(1000);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setPower(0.8);
rightmotor.setPower(0.8);
```

```
while (opModeIsActive()){

    }
```

Agora que todas as três etapas do RUN_TO_POSITION foram adicionadas ao código, o código pode ser testado. No entanto, se você deseja aguardar o motor atingir sua posição alvo antes de continuar no programa, pode usar um loop while que verifica se o motor está ocupado (ainda não atingiu seu destino). Para este programa, vamos editar o while (opModeIsActive()) {}.

> Lembre-se de que, dentro de um Op Mode linear, um loop while deve sempre ter o booleano opModeIsActive() como condição. Essa condição garante que o loop while será encerrado quando o botão de parada for pressionado.

Edite o loop while para incluir as funções leftmotor.isBusy() e rightmotor.isBusy(). Isso verificará se o motor esquerdo e o motor direito estão ocupados se movendo para uma posição alvo. O loop while será interrompido quando qualquer um dos motores atingir a posição alvo.

```
while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {


}
```

> Atualmente, o loop while está esperando que qualquer um dos motores atinja a posição alvo. Pode haver ocasiões em que você deseja aguardar que ambos os motores atinjam sua posição alvo. Nesse caso, o seguinte loop pode ser usado:
>
> ```
> while (opModeIsActive() && (leftmotor.isBusy() || rightmotor.isBusy()))
> ```

> Grave e execute o modo de operação duas vezes seguidas. O robô se move conforme esperado na segunda vez? Tente desligar e depois ligar o Control Hub. Como o robô se move?

Na seção Conceitos Básicos de Encoder, é esclarecido que todas as portas de encoder começam em 0 ticks quando o Control Hub é ligado. Como você não desligou o Control Hub entre as execuções, na segunda vez que você executou o modo de operação, os motores já estavam, ou próximos, à posição alvo. Ao executar um código, é desejável garantir que certas variáveis comecem em um estado conhecido. Para os ticks do encoder, isso pode ser alcançado configurando o modo como STOP_AND_RESET_ENCODER. Adicione este bloco ao modo de operação na seção de inicialização. Cada vez que o modo de operação é inicializado, os ticks do encoder serão resetados para zero.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");


        rightmotor.setDirection(DcMotor.Direction.REVERSE);


        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);


        waitForStart();
```

> Para obter mais informações sobre o modo de motor STOP_AND_RESET_ENCODERS, consulte a seção STOP_AND_RESET_ENCODERS no guia de Uso de Encoders.

Converter Ticks do encoder para uma distância

Na seção anterior, a estrutura básica necessária para usar RUN_TO_POSITION foi criada. A inserção de leftmotor.setTargetPosition(1000); e rightmotor.setTargetPosition(1000); no código define a posição alvo como 1000 ticks. Qual é a distância do ponto inicial do robô até o ponto para o qual o robô se move após a execução deste código?

Em vez de tentar medir ou estimar a distância que o robô percorre, os ticks do encoder podem ser convertidos de quantidade de ticks por revolução do encoder para quantos ticks do encoder são necessários para mover o robô por uma unidade de distância, como milímetros ou polegadas. Saber a quantidade de ticks por unidade de medida permite definir uma distância específica. Por exemplo, se você passar pelo processo de conversão e descobrir que um conjunto de rodas leva 700 ticks para mover uma polegada, isso pode ser usado para calcular o número total de ticks necessários para mover o robô por 24 polegadas.

> Lembre-se de que o guia tem como base a Class Bot V2. O REV DUO Build System utiliza o sistema métrico. Como parte do processo de conversão faz referência ao diâmetro das rodas, esta seção fará a conversão para ticks por milímetro.

Para o processo de conversão as seguintes informações são necessárias:

- Ticks que ocorrem em uma revolução completa do encoder.
- Redução total de engrenagens no motor
  - Incluindo todas as reduções proporcionadas por caixas de engrenagens e componentes de transmissão de movimento, como engrenagens, correntes e coroas,

ou correias e polias.
- Circunferência das rodas

Ticks por revolução

A quantidade de ticks por revolução do eixo do encoder depende do motor e do encoder. Os fabricantes de motores com encoders embutidos fornecerão informações sobre a quantidade de ticks por revolução. Para os Motores HD Hex, o encoder conta com 28 ticks por revolução do eixo do motor.

> Visite o site do fabricante do seu motor ou encoder para obter mais informações sobre a contagem de ticks do encoder. Para os Motores HD Hex ou Motores Core Hex, consulte a documentação do motor em nosso site.

Redução total

Como os ticks por revolução do eixo do encoder ocorrem antes de qualquer redução de engrenagem, é necessário calcular a redução total da engrenagem. Isso inclui a caixa de engrenagens e qualquer redução adicional proveniente dos componentes de transmissão de movimento. Para encontrar a redução total da engrenagem, use a fórmula de Engrenagem Composta.

Para o Class Bot V2, existem duas Carcaças UltraPlanetary, 4:1 e 5:1, e uma redução adicional de engrenagem da Saída UltraPlanetary para as rodas, na proporção de 72 dentes para 45 dentes.

> As UltraPlanetary usam a relação de engrenagem nominal como descritor. As relações de engrenagem reais podem ser encontradas no Manual do Usuário do UltraPlanetary.

Utilizando a fórmula de engrenagens compostas para o robô V2 a redução total é:

3.61/1 * 5.23/1 * 72/45 = 30.21

> Ao contrário das engrenagens helicoidais usadas para transferir movimento para as rodas, as Carcaças de Caixa de Engrenagens UltraPlanetary são sistemas de engrenagens planetárias. Para facilitar os cálculos, as relações de engrenagem para as Carcaças já estão reduzidas.

Circunferência da roda

O Class Bot V2 utiliza as Rodas de Tração de 90mm. 90mm é o diâmetro da roda. Para obter a circunferência apropriada, utilize a seguinte fórmula:

cirncunferência = diãmetro * π

Você pode calcular isso com papel e caneta, mas para os propósitos desse guia, isso pode ser calculado pelo código.

> Devido ao desgaste e tolerâncias de fabricação, o diâmetro de algumas rodas pode ser nominalmente diferente. Para obter resultados mais precisos, considere medir sua roda para confirmar que o diâmetro é preciso.

Para recapitular, para o Classe Robô V2 as seguintes informações são verdadeiras:

Descrição	Valor
Ticks por revolução	28 ticks
Redução total	30.21
Circunferência da roda	90mm * π

Cada uma dessas informações será usada para encontrar o número de ticks do encoder (ou contagens) por milímetro que a roda se move. Em vez de se preocupar em calcular essas informações manualmente, esses valores podem ser adicionados ao código como variáveis constantes. Para fazer isso, crie três variáveis:

- COUNTS_PER_MOTOR_REV
- DRIVE_GEAR_REDUCTION
- WHEEL_CIRCUNFERENCE_MM

> A convenção de nomenclatura comum para variáveis constantes é conhecida como CONSTANT_CASE, onde o nome da variável está todo em maiúsculas e as palavras são separadas por um sublinhado.

Adicione as variáveis à classe OpMode, onde as variáveis de hardware são definidas. Definir as variáveis dentro dos limites da classe, mas fora do modo operacional (op mode), permite que elas sejam referenciadas em outros métodos ou funções dentro da classe. Para garantir que as variáveis sejam referenciáveis, elas são definidas como variáveis static final double. O modificador static permite referências às variáveis em qualquer lugar dentro da classe, e o modificador final indica que essas variáveis são constantes e não são alteradas em outros lugares no código. Como essas variáveis não são do tipo inteiro, são classificadas como variáveis double.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
```

```
    private DcMotor rightmotor;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.21;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * Math.PI;
```

Agora que essas três variáveis foram definidas, elas podem ser usadas para calcular outras duas variáveis: a quantidade de contagens do encoder por rotação da roda e o número de contagens por milímetro que a roda se move.

Para calcular as contagens por revolução da roda, multiplique COUNTS_PER_MOTOR_REV por DRIVE_GEAR_REDUCTION. Use a seguinte fórmula:

$$y = a * b$$

Onde:

- **a = COUNTS_PER_MOTOR_REV**
- **b = DRIVE_GEAR_REDUCTION**
- **y = COUNTS_PER_WHEEL_REV**

Crie a variável **COUNTS_PER_MOTOR_REV** pelo código. Ela também pode ser *static final double*.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION;
```

Uma vez que COUNTS_PER_WHEEL_REV é calculado, use-o para calcular os pulsos por milímetro que a roda se move. Para fazer isso, divida COUNTS_PER_WHEEL_REV pela CIRCUNFERÊNCIA_DA_RODA_MM. Utilize a seguinte fórmula.

$$x = (a * b)/c = y/c$$

Onde:

- **a = COUNTS_PER_MOTOR_REV**
- **b = DRIVE_GEAR_REDUCTION**

- **c = WHEEL_CIRCUMFERENCE_MM**
- **y = COUNTS_PER_WHEEL_REV**
- **x = COUNTS_PER_MM**

Crie a variável **COUNTS_PER_MM** pelo código. Ela também pode ser *static final double*.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotor leftmotor;
    private DcMotor rightmotor;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION;
    static final double     COUNTS_PER_MM           = COUNTS_PER_WHEEL_REV / WHEEL_CIRCUMFERENCE_MM;
```

> COUNTS_PER_WHEEL_REV será criado como uma variável separada de COUNTS_PER_MM, pois é usado no cálculo de uma velocidade alvo.

Movendo para um distância alvo

Agora que você criou as variáveis constantes necessárias para calcular a quantidade de pulsos por milímetro movido, você pode usá-las para definir uma distância alvo. Por exemplo, se você quiser que o robô se mova para frente dois pés, converter de pés para milímetros e multiplicar pelo COUNTS_PER_MM fornecerá a quantidade de pulsos necessários para alcançar essa distância.

Crie mais duas variáveis chamadas leftTarget e rightTarget. Essas variáveis podem ser alteradas e editadas em seu código para indicar às rodas motrizes as posições para as quais devem ir. Em vez de colocá-las com as variáveis constantes, crie essas variáveis dentro do modo de operação (op mode) mas acima do comando waitForStart();.

> A função setTargetPosition(); aceita um tipo de dado inteiro (int) como parâmetro, em vez de um tipo duplo (double). Já que tanto leftTarget quanto rightTarget serão usados para definir a posição alvo, crie ambas as variáveis como variáveis int.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");
```

```
rightmotor.setDirection(DcMotor.Direction.REVERSE);

leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

int leftTarget;
int rightTarget;

waitForStart();
```

Atualmente, o principal fator de distância é COUNTS_PER_MM; no entanto, você pode querer percorrer uma distância que está no sistema imperial, como 2 pés (ou 24 polegadas). A distância alvo, nesse caso, precisará ser convertida para milímetros. Para converter de pés para milímetros, utilize a seguinte fórmula:

d(mm) = d(ft) * 304.8

Se você converter 2 pés para milímetros, o resultado será 609,6 milímetros. Para fins deste guia, arredondemos isso para 610 milímetros. Multiplique 610 milímetros pela variável COUNTS_PER_MM para obter o número de pulsos necessários para mover o robô 2 pés. Como a intenção é fazer com que o robô se mova em linha reta, defina tanto leftTarget quanto rightTarget para serem iguais a 610 * COUNTS_PER_MM.

> Como mencionado anteriormente, a função setTargetPosition(); exige que seu parâmetro seja do tipo de dado inteiro. As variáveis leftTarget e rightTarget foram definidas como inteiros, no entanto, a variável COUNTS_PER_MM é do tipo double. Uma vez que esses são dois tipos de dados diferentes, é necessário fazer uma conversão de tipos de dados. Neste caso, COUNTS_PER_MM precisa ser convertido para um inteiro. Isso é tão simples quanto adicionar a linha (int) na frente da variável double. No entanto, é preciso ter cautela com possíveis erros de arredondamento. Como COUNTS_PER_MM faz parte de uma equação, é recomendável converter para um inteiro após encontrar o resultado da equação. O exemplo de como fazer isso é mostrado abaixo.

```
int leftTarget = (int)(610 * COUNTS_PER_MM);
int rightTarget = (int)(610 * COUNTS_PER_MM);
```

Edite as linhas setTargetPosition(); para que ambos os motores sejam configurados para a posição alvo apropriada. Para fazer isso, adicione as variáveis leftTarget e rightTarget aos seus motores respectivos.

```
leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);
```

> Tente executar o código e observar o comportamento do robô. Considere algumas das seguintes perguntas:
>
> - O robô está se movendo para frente por dois pés?
> - O robô parece estar se movendo em linha reta?
> - O código está sendo executado sem erros?

Definindo a velocidade

A velocidade é um controle em malha fechada dentro do SDK que utiliza as contagens do encoder para determinar a potência/aproximada velocidade necessária para que os motores alcancem a velocidade definida. Ao trabalhar com configurações de encoder, é recomendável definir uma velocidade em vez de um nível de potência, pois oferece um maior controle.

Para definir uma velocidade, é importante entender a velocidade máxima em RPM que o seu motor é capaz de atingir. No caso do Bot Classe V2, os motores são capazes de atingir no máximo 300 RPM. Com um conjunto de rodas motrizes, é provável que você obtenha um melhor controle configurando a velocidade para um valor inferior ao máximo. Neste caso, vamos definir a velocidade para 175 RPM.

> Lembre-se que a função setVelocity é medida em ticks por segundo

Crie uma nova variável *TPS*. Adicione TPS ao Op Mode abaixo de onde rightTarget está definido.

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotor.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotor.class, "rightmotor");


        rightmotor.setDirection(DcMotor.Direction.REVERSE);


        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);


        int leftTarget = (int)(610 * COUNTS_PER_MM);
        int rightTarget = (int)(610 * COUNTS_PER_MM);
        double TPS;


        waitForStart();
```

Como RPM representa a quantidade de rotações por minuto, é necessário fazer uma conversão de RPM para ticks por segundo. Para fazer isso, divida o RPM por 60 para obter a quantidade de rotações por segundo. Em seguida, multiplique as rotações por segundo por COUNTS_PER_WHEEL_REV para obter a quantidade de ticks por segundo.

TPS = 175/69 * CPW R

```
double TPS = (175/60) * COUNTS_PER_WHEEL_REV
```

Substítua as funções **setPower();** para **setVelocity();**. Adicione TPS como parâmetro **setVelocity();**

```
waitForStart();

leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(TPS);
rightmotor.setVelocity(TPS);

while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())){

    }
```

> Tente compilar o código. Você obteve erros?

Com o estado atual do código você vai conseguir os seguintes erros:

```
org/firstinspires/ftc/teamcode/HelloWorld_EncoderAuton.java line 55, column 18: ERROR: cannot find symbol
  symbol:   method setVelocity(double)
  location: variable leftmotor of type com.qualcomm.robotcore.hardware.DcMotor
org/firstinspires/ftc/teamcode/HelloWorld_EncoderAuton.java line 56, column 19: ERROR: cannot find symbol
  symbol:   method setVelocity(double)
  location: variable rightmotor of type com.qualcomm.robotcore.hardware.DcMotor
```

Isso ocorre porque a função setVelocity(); é uma função da interface DcMotorEx. A interface DcMotorEx é uma extensão da interface DcMotor, que fornece funcionalidades avançadas do motor, como acesso a funções de controle em malha fechada. Para utilizar setVelocity();, as variáveis do motor precisam ser alteradas para DcMotorEx. Para fazer isso, tanto a criação das

variáveis privadas dos motores quanto o mapeamento de hardware precisam ser alterados para DcMotorEx.

```
public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotorEx leftmotor;
    private DcMotorEx rightmotor;
```

```
public void runOpMode() {
        leftmotor = hardwareMap.get(DcMotorEx.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotorEx.class, "rightmotor");
```

> Como DcMotorEx é uma extensão de DcMotor, as funções específicas de DcMotor podem ser utilizadas por variáveis definidas como DcMotorEx.

Depois de fazer essas alterações, o código básico para mover dois pés está concluído! O código abaixo é a versão finalizado do código. Nesta versão, outros componentes de hardware e a telemetria foram adicionados.

```
@Autonomous

public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotorEx leftmotor;
    private DcMotorEx rightmotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION;
    static final double     COUNTS_PER_MM           = COUNTS_PER_WHEEL_REV / WHEEL_CIRCUMFERENCE_MM;

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotorEx.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotorEx.class, "rightmotor");
```

```
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);



        int leftTarget = (int)(610 * COUNTS_PER_MM);
        int rightTarget = (int)(610 * COUNTS_PER_MM);
        double TPS = (175/ 60) * COUNTS_PER_WHEEL_REV;

        waitForStart();



        leftmotor.setTargetPosition(leftTarget);
        rightmotor.setTargetPosition(rightTarget);

        leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
        rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

        leftmotor.setVelocity(TPS);
        rightmotor.setVelocity(TPS);

        while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {
            telemetry.addData("left", leftmotor.getCurrentPosition());
            telemetry.addData("right", rightmotor.getCurrentPosition());
            telemetry.update();
        }
    }
}
```

Melhorando a transmissão utilizando RUN_TO_POSITION

Na seção de Navegação do Robô - OnBot Java, foi discutido o mecanismo de setPower();. setPower(); dita em que direção e velocidade um motor se move. Em um drivetrain, isso determina se o robô se move para frente, para trás ou vira.

No modo RUN_TO_POSITION, os contadores do codificador (ou setTargetPosition();) são usados em vez de setPower(); para determinar a direção do motor. Se o valor da posição alvo for maior que a posição atual do codificador, o motor se move para frente. Se o valor da posição alvo for menor que a posição atual do codificador, o motor se move para trás.

Como a velocidade e a direção impactam como um robô vira, setTargetPosition(); e setVelocity(); precisam ser editados para fazer com que o robô vire. Considere o código a seguir:

```
int leftTarget = (int)(610 * COUNTS_PER_MM);
int rightTarget = (int)(-610 * COUNTS_PER_MM);
double TPS = (100/ 60) * COUNTS_PER_WHEEL_REV;



waitForStart();



leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(TPS);
rightmotor.setVelocity(TPS);
```

O rightTarget foi alterado para ser uma posição alvo negativa. Supondo que o codificador comece em zero devido a STOP_AND_RESET_ENCODER, isso faz com que o robô vire para a direita. A velocidade permanece a mesma para ambos os motores. Se você tentar executar este código, pode notar que o robô gira em torno de seu centro de rotação. Para obter uma curva mais ampla, altere a velocidade de modo que o motor direito esteja funcionando a uma velocidade menor do que o motor esquerdo. Ajuste a velocidade e a posição alvo conforme necessário para obter a curva desejada.

> Para obter mais informações sobre como a direção e a velocidade impactam o movimento de um robô, consulte a explicação de setPower(); na seção de Navegação do Robô.

O código a seguir mostra como adicionar uma curva ao programa após o robô se mover para frente por 2 pés. Após o robô atingir a meta de 2 pés, há uma chamada para STOP_AND_RESET_ENCODERS, o que reduzirá a necessidade de calcular qual posição alcançar

após atingir uma posição.

```
@Autonomous

public class HelloWorld_EncoderAuton extends LinearOpMode {
    private DcMotorEx leftmotor;
    private DcMotorEx rightmotor;
    private DcMotor arm;
    private Servo claw;
    private DigitalChannel touch;
    private Gyroscope imu;

    static final double     COUNTS_PER_MOTOR_REV    = 28.0;
    static final double     DRIVE_GEAR_REDUCTION    = 30.24;
    static final double     WHEEL_CIRCUMFERENCE_MM  = 90.0 * 3.14;

    static final double     COUNTS_PER_WHEEL_REV    = COUNTS_PER_MOTOR_REV * DRIVE_GEAR_REDUCTION;
    static final double     COUNTS_PER_MM           = COUNTS_PER_WHEEL_REV / WHEEL_CIRCUMFERENCE_MM;

    @Override
    public void runOpMode() {
        imu = hardwareMap.get(Gyroscope.class, "imu");
        leftmotor = hardwareMap.get(DcMotorEx.class, "leftmotor");
        rightmotor = hardwareMap.get(DcMotorEx.class, "rightmotor");
        arm = hardwareMap.get(DcMotor.class, "arm");
        claw = hardwareMap.get(Servo.class, "claw");
        touch = hardwareMap.get(DigitalChannel.class, "touch");

        rightmotor.setDirection(DcMotor.Direction.REVERSE);

        leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
        rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

        // TPS variable split to change velocity for each motor when necessary

        int leftTarget = (int)(610 * COUNTS_PER_MM);
        int rightTarget = (int)(610 * COUNTS_PER_MM);
        double LTPS = (175/ 60) * COUNTS_PER_WHEEL_REV;
        double RTPS = (175/ 60) * COUNTS_PER_WHEEL_REV;

        waitForStart();
```

```
leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(LTPS);
rightmotor.setVelocity(RTPS);

//wait for motor to reach position before moving on
while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {
    telemetry.addData("left", leftmotor.getCurrentPosition());
    telemetry.addData("right", rightmotor.getCurrentPosition());
    telemetry.update();
}
// Reset encoders to zero
leftmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);
rightmotor.setMode(DcMotor.RunMode.STOP_AND_RESET_ENCODER);

// changing variables to match new needs

leftTarget = (int)(300 * COUNTS_PER_MM);
rightTarget = (int)( -300 * COUNTS_PER_MM);
LTPS = (100/ 60) * COUNTS_PER_WHEEL_REV;
RTPS = (70/ 60) * COUNTS_PER_WHEEL_REV;

leftmotor.setTargetPosition(leftTarget);
rightmotor.setTargetPosition(rightTarget);

leftmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);
rightmotor.setMode(DcMotor.RunMode.RUN_TO_POSITION);

leftmotor.setVelocity(LTPS);
rightmotor.setVelocity(RTPS);

//wait for motor to reach position before moving on
while (opModeIsActive() && (leftmotor.isBusy() && rightmotor.isBusy())) {
    telemetry.addData("left", leftmotor.getCurrentPosition());
    telemetry.addData("right", rightmotor.getCurrentPosition());
```

```
            telemetry.update();
        }
    }
}
```