FIRST ROBOTICS COMPETITION

REEFSCAPE℠ PRESENTED BY HAAS Gene Haas Foundation

FIRST® DIVE℠
presented by Qualcomm

firstinspires.org/robotics/frc

Compétition de robotique *FIRST*® 2025

Guide de programmation Java pour KitBot

1 Contenu

2 Présentation du document

Ce document vous explique comment faire fonctionner votre KitBot 2025 à l'aide de l'exemple de code Java fourni. Pour éviter la duplication du contenu, ce document renvoie fréquemment à la documentation de WPILib et/ou RobotPy pour accomplir certaines étapes spécifiques. En plus de vous permettre d'utiliser le code fourni, ce document présente la structure du programme afin que vous puissiez en comprendre le fonctionnement. Enfin, nous passerons en revue certaines des modifications les plus probables que vous pourriez souhaiter apporter au code et nous fournirons des exemples concrets sur la manière de procéder à ces modifications.

Pour commencer à utiliser l'exemple de code ou pour effectuer certaines des modifications décrites, une compréhension minimale de Java est nécessaire. Le programme et les exemples de modification fournis devraient fournir suffisamment de modèles pour que vous puissiez commencer à travailler. Pour comprendre le guide ou pour apporter des modifications non décrites dans ce document, une connaissance plus approfondie de Java est probablement nécessaire.

Ce document ainsi que l'exemple de code fourni supposent l'utilisation des contrôleurs SPARK MAX fournis dans le kit de démarrage des recrues.

3 Débuter avec le programme pour KitBot

3.1 Câblage de votre robot

Utilisez le document WPILib Zero-to-Robot wiring (voa) pour vous aider à câbler votre robot. Le câblage et le programme du KitBot sont documentés avec les composants du système de contrôle qui ont été récemment ajoutés au kit de pièces pour les recrues (c'est-à-dire les contrôleurs Spark MAX et PDH REV). Le câblage et le code du KitBot peuvent être adaptés à d'autres composants électroniques, mais cette adaptation n'est pas couverte par ces documents.

3.2 Configuration du matériel et de l'environnement de développement

Avant de pouvoir charger le code et tester votre robot, vous devez configurer votre matériel (roboRIO, radio, etc.) et mettre en place votre environnement de développement. Suivez les étapes 2 à 4 du Guide WPILib Zero-to-Robot pour tout mettre en place et vous assurer que vous pouvez déployer un projet de robot de base.

3.3 Ouverture de l'exemple KitBot 2025

L'exemple de code KitBot 2025 est fourni dans des fichiers zip individuels pour chaque langage de programmation. Le code Java contient deux projets complets qui illustrent différentes manières de créer des commandes. Vous trouverez ci-dessous Façons de créer des **commandes** une description de ces différences. Pour ouvrir le code Java :

1. Téléchargez et décompressez l'exemple de code Java. Veillez à décompresser ou à copier dans un emplacement permanent, et non dans un dossier temporaire.
2. Ouvrir **WPILib VS Code** en utilisant le menu Démarrer ou les raccourcis du bureau.
3. En haut à gauche, cliquez sur **File->Open Folder** (Fichier->Ouvrir un dossier) et naviguez jusqu'au dossier "Java" à l'intérieur de l'exemple de code dézippé, puis ouvrez l'un des deux exemples de projet souhaités et cliquez sur **Select Folder** (Sélectionner un dossier).

3.4 Mise à jour du micrologiciel Spark MAX et ID CAN

Avant d'utiliser les SPARK MAX avec le contrôle CAN, il faut leur attribuer à chacun un identifiant unique.

1. Installer le Client matériel REV (REV Hardware Client)
2. Le robot étant hors tension, connectez un câble USB entre l'ordinateur et le port USB du SPARK MAX. Le fait de laisser le robot hors tension permet de s'assurer que seul le SPARK MAX est alimenté et d'éviter de modifier les identifiants de périphériques non désirés.
3. Mettre à jour le micrologiciel du SPARK MAX
4. Réglez l'ID CAN et le type de moteur (vous pouvez ignorer la limite de courant) et enregistrez les paramètres.

a. Les identifiants CAN de chaque appareil se trouvent dans Constants.java. Vous pouvez soit régler les dispositifs pour qu'ils correspondent à ces ID, soit régler les ID comme vous le souhaitez (certaines équipes règlent l'ID CAN = le numéro de canal auquel le dispositif est attaché sur le PD) et ensuite mettre à jour ces constantes.
b. Note : Si vous souhaitez faire tourner le moteur comme décrit sur cette page, assurez-vous que le robot est dans un état sûr pour le faire (les roues ne touchent pas le sol ou la table, les mains sont à l'écart du robot).

5. Répétez l'opération pour les 5 appareils du robot. Si vous avez câblé le 6e Spark MAX, vous voudrez probablement lui attribuer également une ID non conflictuelle.
6. Bien que cela ne soit pas obligatoire, si vous utilisez le PDH REV, vous pouvez également vérifier qu'il dispose du dernier micrologiciel à ce moment-là. Ne modifiez pas l'ID du PDH par rapport à la valeur par défaut, chaque type d'appareil dispose d'un espace ID distinct et votre PDH n'entrera pas en conflit avec votre SPARK MAX même s'il est réglé sur la même ID.

Maintenant que tous vos appareils sont configurés, vous pouvez effectuer une vérification préliminaire du câblage de votre bus CAN à l'aide du Client matériel REV. Lorsque vous êtes connecté à un appareil REV sur votre bus CAN à l'aide d'un câble USB, mettez le robot sous tension et vous devriez voir tous les autres appareils répertoriés dans le volet gauche du Client matériel REV, sous l'intitulé CAN Bus. Si vous ne voyez pas tous les dispositifs, il est probable que vous ayez un ou plusieurs problèmes avec le câblage de votre bus CAN :

1. Vérifiez que votre bus CAN commence par le roboRIO et se termine par une résistance de 120 ohms, ou par le terminateur intégré d'un concentrateur de distribution d'énergie (Power Distribution Hub) ou d'un panneau de distribution d'énergie (Power Distribution Panel) (la terminaison étant réglée sur On à l'aide du cavalier ou de l'interrupteur approprié).
2. Vérifiez que les connexions de votre bus CAN correspondent toutes à jaune-jaune et à vert-vert.
3. Vérifiez que toutes les connexions des fils CAN sont bien fixées les unes aux autres et que les connecteurs sont bien installés dans chaque SPARK Max.
4. Si le problème persiste, déplacez la connexion USB sur différents appareils et observez ce que chaque appareil peut "voir" sur le bus; ceci peut aider à localiser le problème.

3.5 Installation de REVLib

La bibliothèque logicielle pour le SPARK MAX en mode CAN est fournie par le fournisseur (REV Robotics). La configuration de la bibliothèque tierce est déjà incluse dans le projet, mais vous devrez installer la bibliothèque elle-même. Il y a deux façons de procéder :

1. **Recommandé** - Installer la bibliothèque hors ligne - Cela garantira que la bibliothèque persiste sur votre machine même si vous ne construisez pas de nouveau code pendant un certain temps (les installations en ligne peuvent être nettoyées automatiquement par Gradle).
    a. Téléchargez la dernière version de REVLib en utilisant le lien de la documentation REV.
    b. Décompressez dans le répertoire *C:\Users\Public\wpilib\2025* sous Windows et dans le répertoire *~/wpilib/2025* sur les systèmes de type Unix.

2. Installation en ligne - Lorsque l'ordinateur est connecté à Internet, cliquez sur l'icône WPILib en haut à droite de la fenêtre VSCode pour faire apparaître l'invite de l'extension WPILib, puis commencez à taper "Build Robot Code" et sélectionnez cette option lorsqu'elle apparaît. La configuration de la bibliothèque étant déjà incluse dans le projet, la bibliothèque sera automatiquement récupérée sur Internet.

Pour installer des bibliothèques de fournisseurs supplémentaires en ligne, vous pouvez utiliser le gestionnaire de bibliothèques de fournisseurs WPILib dans VS Code.

3.6 Déployer et tester l'exemple KitBot

Pour déployer l'exemple sur votre robot, vous devez définir le numéro d'équipe dans le projet. Cliquez sur l'**icône WPILib** dans le coin supérieur droit de la fenêtre VS Code (Logo qui ressemble à un symbole rouge et gris '<<>>') pour ouvrir l'invite WPILib et commencez à taper "Set Team Number" et sélectionnez cette option lorsqu'elle apparaît. Saisissez votre numéro d'équipe (sans 0 à gauche - par exemple 123 ou 9996) et appuyez sur Entrée.

Vous êtes maintenant prêt à déployer l'exemple KitBot de la même manière que vous avez déployé le projet test à l'étape 4 du guide Zero-to-Robot.

Avertissement : Assurez-vous d'avoir de l'espace dans toutes les directions lorsque vous utilisez un robot. Même avec un programme connu, le robot peut se déplacer à une vitesse ou dans une direction inattendue. Soyez prêt à désactiver (Entrée) ou à arrêter le robot d'urgence (Barre d'espacement) si nécessaire. Le code 2025 du KitBot contient une routine autonome très simple qui déplace le robot vers l'avant à une vitesse de 50% pendant 1 seconde lorsque le robot est activé en mode autonome.

3.7 Configuration des manettes de jeu

Le code est configuré pour utiliser la classe de contrôleur Xbox. Les manettes de jeu Logitech F310 fournies dans le kit de pièces apparaîtront comme des manettes Xbox pour le logiciel WPILib si elles sont configurées dans le bon mode. Pour configurer les contrôleurs, vérifiez que le commutateur situé à l'arrière du contrôleur est réglé sur la position "X". Ensuite, lorsque vous utilisez le contrôleur, assurez-vous que la DEL située à côté du bouton Mode est éteinte ; si elle est allumée, appuyez sur le bouton Mode pour la faire basculer. Lorsque le voyant Mode est allumé, la manette permute la fonction du stick analogique gauche et du D-pad.

3.8 Que fait le code ?

Le programme fourni met en œuvre les commandes du robot suivantes dans Teleoperated :

- La manette de commande est une manette Xbox dans l'emplacement 0 de la station de pilotage.
    - Contrôle le groupe motopropulseur du robot à l'aide de la conduite Arcade Split-stick
        - L'axe Y (vertical) du stick gauche contrôle le mouvement avant-arrière du groupe motopropulseur.
        - L'axe X (horizontal) du stick droit contrôle la rotation du groupe motopropulseur.

- La manette de l'opérateur est une manette Xbox dans l'emplacement 1 de la station de pilotage.
    - Contrôle le rouleau des pièces de jeu à l'aide des gâchettes et des boutons
        - Gâchette gauche - Fait tourner le moteur du rouleau vers l'intérieur (repousse le Corail vers le haut de la rampe) à vitesse variable.
        - Déclencheur droit - Fait fonctionner le moteur du rouleau vers l'extérieur (éjecte le corail) à vitesse variable.
        - Bouton A - Fait fonctionner le moteur du rouleau vers l'extérieur à une puissance spécifique.

En raison de la manière dont le code est établi, il n'est pas recommandé d'appuyer sur les déclencheurs en même temps, car le comportement sera difficile à prévoir. Le fait d'appuyer simultanément sur les gâchettes gauche et droite permet d'additionner leurs valeurs. Par exemple, en appuyant à moitié sur la gâchette gauche et à fond sur la gâchette droite, le moteur tournera vers l'extérieur à 50% de vitesse.

4 Structure générale du programme

Le code fourni utilise la structure de programmation basée sur les commandes fournie par WPILib. Cette structure divise les actionneurs du robot en "sous-systèmes" qui sont contrôlés par des "commandes" ou des collections de commandes (appelées à juste titre "groupes de commandes"). La structure basée sur les commandes peut être un peu excessive pour un robot de cette complexité, mais elle s'adapte très bien aux équipes qui cherchent à ajouter des fonctionnalités supplémentaires à leur KitBot. En outre, cette structure de code a été utilisée par plus de 60 % des équipes en 2024, ce qui augmente la probabilité que les équipes autour de vous soient en mesure de fournir une assistance avant ou pendant l'événement.

Pour en savoir plus sur la structure basée sur les commandes, voir le chapitre Programmation basée sur les commandes de la documentation WPILib.

4.1 Façons de créer des commandes

Il existe plusieurs façons de définir des commandes dans la structure basée sur les commandes. Ce projet utilise deux de ces types pour donner une idée de ce à quoi ils ressembleraient dans un projet de robot complet. Les méthodes courantes de création de commandes utilisées dans ce projet sont les suivantes :

- Traditionnel : La commande/groupe est défini.e comme sa propre classe dans son propre fichier.
- Manufacture : La commande/groupe est défini.e par une "méthode de manufacture de commande" dans le sous-système ou par une classe de commande statique distincte.

Traditionnel	Manufacture
Généralement plus facile à comprendre	**Courbe d'apprentissage légèrement élevée**
Modularité - Les commandes peuvent être longues en fonction de la complexité.	**Modularité - Les commandes sont décomposées en petits éléments et rassemblées en groupes.**
Standards - Les classes de commande nécessitent une sous-classification et peuvent être longues.	**Standards - Les commandes sont écrites comme des méthodes dans le sous-système, ce qui permet de réduire le code inutile.**
Organisation - Le fait d'avoir de nombreuses classes de commandes peut entraîner un certain désordre et ralentir l'efficacité du programmeur.	**Organisation - Les commandes sont regroupées en fonction des sous-systèmes qu'elles requièrent, ce qui permet de réduire le nombre de classes de commandes dédiées ou de les supprimer.**

Débogage - Plus facile à déboguer en raison d'une structure mieux définie et d'une logique traditionnelle.	**Débogage - Plus difficile à déboguer en raison des fonctions lambda et des compositions de commandes.**

5 Explication du code

5.1 Sous-systèmes

Comme indiqué dans Qu'est-ce que la programmation basée sur les commandes , les "sous-systèmes représentent des ensembles de matériel robotique contrôlés indépendamment (tels que des contrôleurs de moteur, des capteurs, des actionneurs pneumatiques, etc)".

Pour le KitBot 2025, nous disposons de 5 moteurs qui constituent deux sous-systèmes : le groupe motopropulseur et le rouleau. Les 4 moteurs du groupe motopropulseur doivent toujours fonctionner ensemble pour déplacer le robot sur le terrain et le moteur du rouleau doit tourner pour manipuler les tuyaux.

Parfois, les limites entre les sous-systèmes ne sont pas si claires. Si vous avez un bras avec une épaule et une articulation du poignet et un ensemble de roues motorisées à l'extrémité, s'agit-il d'un seul sous-système ou de plusieurs ? La règle générale à suivre est de réfléchir aux actions, ou commandes, que vous pourriez avoir pour contrôler les sous-systèmes. Pensez-vous que les deux pièces puissent être contrôlées indépendamment l'une de l'autre (par exemple, faire entrer ou sortir l'admission tout en bougeant le bras ou le poignet) ? Si vous n'êtes pas sûr, optez pour des sous-systèmes plus petits ; vous pouvez toujours créer des commandes qui nécessitent plusieurs sous-systèmes, mais si vous souhaitez que des commandes distinctes contrôlent un seul sous-système en même temps, vous devrez remanier le sous-système pour le diviser.

5.1.1 CANDriveSubsystem

Cette classe est le sous-système du groupe motopropulseur.

5.1.1.1 Importations

```
import com.revrobotics.spark.SparkBase.PersistMode;
import com.revrobotics.spark.SparkBase.ResetMode;
import com.revrobotics.spark.SparkLowLevel.MotorType;
import com.revrobotics.spark.SparkMax;
import com.revrobotics.spark.config.SparkMaxConfig;

import edu.wpi.first.wpilibj.drive.DifferentialDrive;
import edu.wpi.first.wpilibj2.command.SubsystemBase;
import frc.robot.Constants.DriveConstants;
```

Cette section déclare les autres classes ou paquets auxquels nous devons faire référence dans ce code (importations). Une pratique courante consiste à ajouter les importations au fur et à mesure ; lorsque vous faites référence à une classe que vous n'avez pas encore importée, vous pouvez ajouter une importation pour cette classe. Le premier groupe d'importations dans ce sous-système provient de la bibliothèque REV Robotics pour divers éléments que nous devons référencer pour les Spark MAX. Le

deuxième groupe est constitué de classes WPILib (une pour le type de groupe motopropulseur du robot et une pour les sous-systèmes) et de la section DriveConstants du fichier Constants de ce projet.

Si vous utilisez la version InlineCommands du projet, vous verrez quelques importations supplémentaires pour certaines classes de commandes WPILib ainsi qu'une en haut pour une classe Java appelée "DoubleSupplier" qui est la façon dont vous passerez les valeurs changeantes dans les commandes.

5.1.1.2 Déclaration de la classe et variables membres

```
public class CANDriveSubsystem extends SubsystemBase {
  private final SparkMax leftLeader;
  private final SparkMax leftFollower;
  private final SparkMax rightLeader;
  private final SparkMax rightFollower;

  private final DifferentialDrive drive;
```

La première ligne est la déclaration de la classe. Ceci déclare le nom de notre classe et indique qu'il s'agit d'une extension de la classe SubsystemBase. Tous les sous-systèmes doivent étendre cette classe qui fournit quelques fonctions utilitaires concernant la définition du nom du sous-système, son enregistrement auprès du planificateur (scheduler) et l'envoi d'informations sur le sous-système au tableau de bord.

5.1.1.3 Constructeur

```
/** Class to drive the robot over CAN */
public CANDriveSubsystem() {
  // create brushed motors for drive
  leftLeader = new SparkMax(DriveConstants.LEFT_LEADER_ID, MotorType.kBrushed);
  leftFollower = new SparkMax(DriveConstants.LEFT_FOLLOWER_ID, MotorType.kBrushed);
  rightLeader = new SparkMax(DriveConstants.RIGHT_LEADER_ID, MotorType.kBrushed);
  rightFollower = new SparkMax(DriveConstants.RIGHT_FOLLOWER_ID, MotorType.kBrushed);

  // set up differential drive class
  drive = new DifferentialDrive(leftLeader, rightLeader);
```

La section suivante est le constructeur. Dans la première partie du constructeur, nous initialisons toutes les variables contenues dans le sous-système. Dans le cas de notre groupe motopropulseur, nous initialisons les 4 contrôleurs de moteur et ajoutons un contrôleur pour chaque côté dans un objet WPILib DifferentialDrive qui décrit l'ensemble de la propulsion.

Le reste du constructeur (non illustré) met en place les SparkMAX pour le groupe motopropulseur. Un moteur de chaque côté est défini comme suiveur et instruit pour suivre le leader, puis les moteurs leader sont configurés avec quelques détails supplémentaires. Examinez les commentaires pour savoir quels paramètres sont configurés et pourquoi.

5.1.1.4 Méthodes

```
38    public void driveArcade(double xSpeed, double zRotation) {
39      drive.arcadeDrive(xSpeed, zRotation);
40    }
```

Le reste de la classe du sous-système est constitué de méthodes. Nous définissons ici toutes les méthodes que les commandes peuvent avoir besoin d'appeler pour obtenir l'état d'un sous-système ou pour effectuer des actions sur celui-ci. Pour notre groupe motopropulseur simple, la seule méthode dont nous avons besoin est la méthode arcadeDrive qui transmet simplement les paramètres à la même méthode sur l'objet DifferentialDrive.

Dans la version InlineCommands du projet, cette méthode sera un peu différente, pour plus de détails voir la section Manufacture de Commandes.

5.1.2 CANRollerSubsystem

Cette classe est le sous-système du mécanisme à rouleaux.

5.1.2.1 Paquet, importations, déclaration de classe, variables membres et constructeur

Les premières sections de ce sous-système sont très similaires à celles du sous-système de déplacement Drive. Voir cette section pour une description plus détaillée de chacune des parties du code. Pour le lanceur, le moteur unique est créé et contrôlé indépendamment, aucun objet de suivi ou de propulsion n'est utilisé.

5.1.2.2 Méthodes - Contrôle du matériel

```
21    public void runRoller(double forward, double reverse) {
22      rollerMotor.set(forward - reverse);
23    }
```

Le reste de la classe du sous-système est constitué de méthodes d'accès au matériel. Nous définissons ici toutes les méthodes que les commandes peuvent avoir besoin d'appeler pour obtenir l'état d'un sous-système ou pour effectuer des actions sur celui-ci. Pour notre rouleau, cela inclut des méthodes pour définir la vitesse de l'essieu.

Dans la version InlineCommands du projet, cette méthode sera un peu différente, pour plus de détails voir la section Manufacture de Commandes.

5.2 Commandes

Les commandes indiquent au robot quand exécuter les différents composants définis dans leurs sous-systèmes. Les commandes sont "planifiées" pour être exécutées, puis supprimées du planificateur de commandes. Chaque commande comporte quatre parties distinctes qui sont exécutées tout au long de son cycle de vie :

- Initialize : exécuté lorsque la commande est initialement planifiée. Tout ce qui est placé dans la section d'initialisation d'une commande sera exécuté juste avant le corps principal de la commande.
- Execute : le corps principal de la commande, qui s'exécute une fois par cycle (~20 ms).

- End : cette opération est effectuée lorsque la commande est retirée du planificateur, ce qui se produit lorsque la commande indique qu'elle est terminée ou si elle est interrompue par une nouvelle commande nécessitant l'un des mêmes sous-systèmes.
- isFinished : appelée une fois par cycle après la méthode execute. isFinished renvoie un résultat positif lorsque la condition de sortie de la commande est remplie. Lorsque isFinished renvoie un résultat positif, la méthode end est appelée.

Comme nous l'avons vu dans la section 4, les deux façons d'écrire des commandes sur lesquelles nous nous concentrons dans ce tutoriel sont les commandes traditionnelles, qui sous-classent la classe Command WPILib, et les manufactures de commandes, qui utilisent des décorateurs de commandes pour créer des commandes. Quelle que soit la manière dont votre équipe choisit de formuler les commandes, c'est la structure sous-jacente que suivent les commandes.

5.3 Commandes traditionnelles

Cette sous-section se concentre sur le cadre traditionnel basé sur les commandes. Les commandes traditionnelles sont définies dans leurs propres classes en sous-classant la Commande générique WPILib. Cela signifie que nous pouvons directement modifier le comportement des méthodes de commande.

5.3.1 DriveCommand

5.3.1.1 Importations et constructeurs

```java
package frc.robot.commands;

import edu.wpi.first.wpilibj2.command.Command;
import frc.robot.subsystems.CANDriveSubsystem;
import java.util.function.DoubleSupplier;

// Command to drive the robot with joystick inputs
public class DriveCommand extends Command {
  private final DoubleSupplier xSpeed;
  private final DoubleSupplier zRotation;
  private final CANDriveSubsystem driveSubsystem;

  // Constructor. Runs only once when the command is first created.
  public DriveCommand(
      DoubleSupplier xSpeed, DoubleSupplier zRotation, CANDriveSubsystem driveSubsystem) {
    // Save parameters to local variables for use later
    this.xSpeed = xSpeed;
    this.zRotation = zRotation;
    this.driveSubsystem = driveSubsystem;

    // Declare subsystems required by this command. This should include any
    // subsystem this command sets and output of
    addRequirements(this.driveSubsystem);
  }
```

Le constructeur de la commande stocke simplement certains paramètres dans des variables de classe en vue d'une utilisation ultérieure et déclare le sous-système dont la commande a besoin en utilisant "addRequirements". Toutes les commandes doivent avoir un "addRequirements" indiquant tous les sous-systèmes sur lesquels elles appelleront des méthodes qui contrôlent les sorties (c'est-à-dire que vous pouvez appeler des méthodes pour obtenir **get** des valeurs d'un sous-système sans l'exiger, mais vous ne devez pas fixer **set** de valeurs).

5.3.1.2 Méthodes - Substituts d'état de commande

```
38    @Override
39    public void initialize() {}
40
41    @Override
42    public void execute() {
43      driveSubsystem.driveArcade(xSpeed.getAsDouble(), zRotation.getAsDouble());
44    }
45
46    @Override
47    public void end(boolean isInterrupted) {}
48
49    @Override
50    public boolean isFinished() {
51      return false;
52    }
```

Dans la méthode execute, nous appelons la méthode ArcadeDrive du sous-système de propulsion qui demande aux moteurs de fonctionner de manière à ce que le robot se déplace en fonction des entrées du joystick. Si vous n'avez pas de code dans une méthode de commande particulière (comme "initialize" ou "end", ici), il est permis de les supprimer, la classe Command de base comprend une implémentation vide qui sera utilisée si votre classe n'a pas de substitut.

5.3.2 RollerCommand

La première section de cette commande est très similaire à la commande Drive. Voir la section **Error! Reference source not found.** 1 pour une description plus détaillée de chacune des parties du code.

5.3.2.1 Importations et constructeurs

```java
package frc.robot.commands;

import edu.wpi.first.wpilibj2.command.Command;
import frc.robot.subsystems.CANRollerSubsystem;
import java.util.function.DoubleSupplier;

// Command to run the roller with joystick inputs
public class RollerCommand extends Command {
  private final DoubleSupplier forward;
  private final DoubleSupplier reverse;
  // private final CANRollerSubsystem rollerSubsystem;
  private final CANRollerSubsystem rollerSubsystem;

  public RollerCommand(
      DoubleSupplier forward, DoubleSupplier reverse, CANRollerSubsystem rollerSubsystem) {
    this.forward = forward;
    this.reverse = reverse;
    this.rollerSubsystem = rollerSubsystem;

    addRequirements(this.rollerSubsystem);
  }
```

5.3.2.2 Méthodes - Substituts d'état de commande

```java
38    @Override
39    public void initialize() {}
40
41    @Override
42    public void execute() {
43      rollerSubsystem.runRoller(forward.getAsDouble(), reverse.getAsDouble());
44    }
45
46    @Override
47    public void end(boolean isInterrupted) {}
48
49    @Override
50    public boolean isFinished() {
51      return false;
52    }
```

Dans la commande execute() du rouleau, le rouleau est réglé sur les valeurs des fournisseurs qui ont été transmises. Le programme fourni utilise cette commande de deux manières : comme commande par défaut, où les valeurs sont récupérées à partir des déclencheurs, et comme commande associée à un bouton, où les valeurs sont fournies sous forme de constantes. Notez que la méthode end() est

vide, ce qui signifie que la commande n'arrête pas explicitement le moteur lorsqu'elle se termine. Cela fonctionne dans le programme fourni car dès que la version bouton se termine, la version commande par défaut commence à fonctionner et arrête les moteurs si les gâchettes ne sont pas actionnées. Si vous utilisez cette commande en Auto, vous remarquerez peut-être que le rouleau ne s'arrête pas comme prévu; ajouter un peu de code à la méthode end nettoiera cela !

5.3.3 AutoCommand

AutoCommand est légèrement différente des deux autres commandes, car elle nécessite un peu plus de configuration.

5.3.3.1 Importations et constructeurs

```
package frc.robot.commands;

import edu.wpi.first.wpilibj.Timer;
import edu.wpi.first.wpilibj2.command.Command;
import frc.robot.subsystems.CANDriveSubsystem;

// Command to run the robot at 1/2 power for 1 second in autonomous
public class AutoCommand extends Command {
  CANDriveSubsystem driveSubsystem;
  private Timer timer;
  private double seconds = 1.0;

  // Constructor. Runs only once when the command is first created.
  public AutoCommand(CANDriveSubsystem driveSubsystem) {
    // Save parameter for use later and initialize timer object.
    this.driveSubsystem = driveSubsystem;
    timer = new Timer();

    // Declare subsystems required by this command. This should include any
    // subsystem this command sets and output of
    addRequirements(driveSubsystem);
  }
```

Le constructeur de AutoCommand prend un paramètre du sous-système de propulsion et instancie deux variables membres pour suivre l'état de la routine auto.

5.3.3.2 Méthodes - Substituts d'état de commande

```
// Runs each time the command is scheduled. For this command, we handle starting
// the timer.
@Override
public void initialize() {
  // start timer, uses restart to clear the timer as well in case this command has
  // already been run before
  timer.restart();
}

// Runs every cycle while the command is scheduled (~50 times per second)
@Override
public void execute() {
  // drive at 1/2 speed
  driveSubsystem.driveArcade(xSpeed:0.5, zRotation:0.0);
}

// Runs each time the command ends via isFinished or being interrupted.
@Override
public void end(boolean isInterrupted) {
  // stop drive motors
  driveSubsystem.driveArcade(xSpeed:0.0, zRotation:0.0);
}

// Runs every cycle while the command is scheduled to check if the command is
// finished
@Override
public boolean isFinished() {
  // check if timer exceeds seconds, when it has this will return true indicating
  // this command is finished
  return timer.get() >= seconds;
```

- Initialize : Démarrer la minuterie.
- Execute : Règle les moteurs de la propulsion pour avancer.
- End : Arrête la minuterie et le sous-système de propulsion.
- IsFinished : Retourne vrai lorsque la minuterie dépasse 1 seconde.

5.4 Manufacture de Commandes

Cette sous-section se concentre sur le cadre "manufacture" basé sur les commandes. Les commandes sont définies dans leurs sous-systèmes à l'aide des décorateurs et des commandes en ligne WPILib.

5.4.1 Manufacture de la Commande Drive

La manufacture de commande de propulsion (Drive) est une méthode du sous-système Drive qui crée une commande. Nous appelons le RobotContainer pour qu'il devienne la commande par défaut du sous-système de propulsion.

```
64    public Command driveArcade(
65        CANDriveSubsystem driveSubsystem, DoubleSupplier xSpeed, DoubleSupplier zRotation) {
66      return Commands.run(
67          () -> drive.arcadeDrive(xSpeed.getAsDouble(), zRotation.getAsDouble()), driveSubsystem);
68    }
```

5.4.2 Manufacture de la commande de rouleaux

Cette section est très similaire à la section Manufacture de la commande Drive. Pour plus de détails sur le fonctionnement des manufactures à base de commandes, voir la section 5.4.1.

```
30    public Command runRoller(
31        CANRollerSubsystem rollerSubsystem, DoubleSupplier forward, DoubleSupplier reverse) {
32      return Commands.run(
33          () -> rollerMotor.set(forward.getAsDouble() - reverse.getAsDouble()), rollerSubsystem);
34    }
```

5.4.3 Autos

Le fichier Autos est un exemple de "manufacture de commandes statiques". Votre programme ne devrait jamais créer d'objet Autos (comme le montre le fait que le constructeur affiche simplement un message d'erreur). Au lieu de cela, vous appelez les méthodes de classe de manière statique en utilisant la syntaxe de type Autos.exampleAuto(). Cette structure est l'une des façons de définir des groupes complexes qui impliquent plusieurs sous-systèmes (bien que notre exemple ici ne soit pas complexe et ne nécessite qu'un seul sous-système).

```
10  public final class Autos {
11    // Example autonomous command which drives forward for 1 second.
12    public static final Command exampleAuto(CANDriveSubsystem driveSubsystem) {
13      return driveSubsystem.driveArcade(driveSubsystem, () -> 0.5, () -> 0.0).withTimeout(seconds:1.0);
14    }
15  }
```

Notre fichier exemple ne comporte qu'une seule routine autonome à obtenir. Vous pourriez facilement étendre ce modèle en ajoutant des méthodes supplémentaires pour définir plus de routines autonomes et vous pourriez choisir entre elles à l'aide d'un SendableChooser sur le tableau de bord.

Cette simple routine autonome demande au robot de rouler en avant pendant 1 seconde à une puissance de 50 % en utilisant le décorateur WithTimeout pour définir un délai d'attente de 1 seconde pour la commande de déplacement. Les différents types de compositions de commandes intégrées via des décorateurs et des méthodes manufactures sont décrits sur la page Compositions des commandes.

5.5 Constantes

Cette classe contient des constantes nommées utilisées ailleurs dans le code. Les sous-classes sont utilisées pour organiser les constantes en groupes distincts, dans ce cas par sous-système. Les noms de constantes fournis devraient décrire assez clairement l'utilité de chacune d'entre elles.

5.6 Robot

Ce fichier est identique au modèle par défaut basé sur les commandes. Vous trouverez une description des éléments de la classe Robot dans Structurer un projet de robot à base de commandes.

5.7 RobotContainer

La classe RobotContainer est l'endroit où sont déclarées les instances des sous-systèmes et des contrôleurs du robot et où sont définies les commandes par défaut et les correspondances entre les boutons et les commandes.

5.7.1 Importations

```
package frc.robot;

import edu.wpi.first.wpilibj.smartdashboard.SendableChooser;
import edu.wpi.first.wpilibj2.command.Command;
import edu.wpi.first.wpilibj2.command.button.CommandXboxController;
import edu.wpi.first.wpilibj2.command.button.Trigger;
import frc.robot.Constants.OperatorConstants;
import frc.robot.Constants.RollerConstants;
import frc.robot.commands.AutoCommand;
import frc.robot.commands.DriveCommand;
import frc.robot.commands.RollerCommand;
import frc.robot.subsystems.CANDriveSubsystem;
import frc.robot.subsystems.CANRollerSubsystem;
```

La première section du code est celle des importations. Dans ce cas, nous devons importer certains éléments du module de commandes, le fichier de constantes de notre projet, puis toutes nos commandes et sous-systèmes.

5.7.2 Définition de la classe et constructeur

```
public class RobotContainer {
  // The robot's subsystems
  private final CANDriveSubsystem driveSubsystem = new CANDriveSubsystem();
  private final CANRollerSubsystem rollerSubsystem = new CANRollerSubsystem();

  // The driver's controller
  private final CommandXboxController driverController = new CommandXboxController(
      OperatorConstants.DRIVER_CONTROLLER_PORT);

  // The operator's controller
  private final CommandXboxController operatorController = new CommandXboxController(
      OperatorConstants.OPERATOR_CONTROLLER_PORT);

  // The autonomous chooser
  private final SendableChooser<Command> autoChooser = new SendableChooser<>();
```

La première section de la classe définit certaines variables membres de la classe. Pour RobotContainer, il s'agit généralement de tous les sous-systèmes et dispositifs de contrôle. Ce code utilise la classe CommandXboxController pour représenter les manettes de jeu, car elle contient un certain nombre de méthodes d'aide qui facilitent grandement la connexion des commandes aux boutons.

```
/**
 * The container for the robot. Contains subsystems, OI devices, and commands.
 */
public RobotContainer() {
  // Set up command bindings
  configureBindings();

  // Set the options to show up in the Dashboard for selecting auto modes. If you
  // add additional auto modes you can add additional lines here with
  // autoChooser.addOption
  autoChooser.setDefaultOption(name:"Autonomous", new AutoCommand(driveSubsystem));
}
```

Vient ensuite le constructeur qui contient un appel à configureBindings() dont nous parlerons plus loin. Cette méthode est utilisée pour définir les liens entre les boutons et les commandes par défaut. Vous pourriez placer tout ce code directement dans le constructeur, mais au fur et à mesure que votre robot et vos commandes deviennent plus complexes, il est souvent utile de séparer les choses pour plus de clarté.

L'autre chose que fait le constructeur est d'ajouter le mode autonome au sélecteur du tableau de bord. Des options supplémentaires de mode autonome peuvent être ajoutées à ce sélecteur à l'aide de la

méthode adoption() et vous pouvez alors sélectionner celui que vous souhaitez exécuter à partir de SmartDashboard, Shuffleboard, Elastic ou d'autres tableaux de bord [tiers].

5.7.3 configureBindings()

Cette méthode établit les relations entre nos contrôles et nos commandes.

```
private void configureBindings() {
  // Set the A button to run the "RollerCommand" command with a fixed
  // value ejecting the gamepiece while the button is held
  operatorController.a()
      .whileTrue(new RollerCommand(() -> RollerConstants.ROLLER_EJECT_VALUE, () -> 0, rollerSubsystem));
```

La première section établit une liaison avec le bouton "A" du contrôleur de l'opérateur. La classe CommandXboxController contient des méthodes pour chaque bouton qui renvoient des objets "Trigger". Ces objets "Trigger" ont ensuite d'autres méthodes qui restreignent le comportement que nous voulons contrôler, comme les bascules, le déclenchement sur changement, ou l'exécution uniquement lorsque le Trigger est vrai ou faux. Dans ce cas, nous utilisons "whileTrue()" pour que notre commande s'exécute pendant que le bouton est maintenu et s'arrête lorsqu'il est relâché. Parce que notre RollerCommand prend des types DoubleSupplier comme paramètres (afin qu'il puisse récupérer les valeurs changeantes du joystick dans l'autre utilisation), nous devons utiliser une fonction lambda pour passer les valeurs constantes.

```
driveSubsystem.setDefaultCommand(new DriveCommand(
    () -> -driverController.getLeftY(), () -> -driverController.getRightX(), driveSubsystem));
```

Le code fourni définit également les commandes par défaut dans la méthode configureBindings. L'endroit où faire cela est une question de préférence personnelle. Au fur et à mesure que votre robot et ses commandes deviennent plus complexes, vous pouvez préférer placer le mappage des commandes par défaut dans le constructeur ou même l'extraire en une méthode propre appelée à partir du constructeur.

Ensuite, nous configurons la commande par défaut pour le groupe motopropulseur. Nous voulons qu'une commande soit exécutée sur notre groupe motopropulseur pour nous permettre de piloter le robot avec des joysticks dès que nous n'avons pas d'autre commande utilisant le groupe motopropulseur (comme la commande exempleAuto). Pour ce faire, nous utilisons la méthode setDefaultCommand() du sous-système. Elle définit la commande qui sera exécutée chaque fois que le planificateur ne verra rien d'autre s'exécuter sur ce sous-système.

Nous utilisons notre DriveCommand et lui transmettons à nouveau des données à l'aide de lambdas, mais dans ce cas, ces données sont des fonctions du driverController que la commande appellera chaque fois qu'elle aura besoin de nouvelles données. Pour le mouvement avant/arrière, nous transmettons la valeur de l'axe Y (vertical) du stick gauche de la manette, mais nous l'inversons. En effet, les joysticks considèrent généralement comme négatif la poussée du stick vers l'extérieur et comme positif de le tirer vers soi (ce qui s'explique par le fait qu'ils étaient utilisés à l'origine pour les simulateurs de vol). Nous voulons que le fait de pousser le stick fasse avancer le robot, donc nous inversons la valeur. Il en va de même pour la valeur de rotation, où l'on inverse l'axe X (horizontal) du stick droit de la manette. Le joystick considère que pousser ce bouton vers la droite est une valeur

positive, mais les classes WPILib considèrent que la rotation dans le sens des aiguilles d'une montre (ce à quoi on s'attendrait en poussant le joystick vers la droite) est une valeur négative.

La liaison pour le sous-système "rouleau" se présente comme suit :

```
rollerSubsystem.setDefaultCommand(new RollerCommand(
    () -> driverController.getRightTriggerAxis(),
    () -> driverController.getLeftTriggerAxis(),
    rollerSubsystem));
```

Ou

```
rollerSubsystem.setDefaultCommand(
    rollerSubsystem.runRoller(
        rollerSubsystem,
        () -> operatorController.getRightTriggerAxis(),
        () -> operatorController.getLeftTriggerAxis()));
```

Cela permet à la commande du rouleau d'obtenir sa valeur avant à partir de la gâchette droite et sa valeur arrière à partir de la gâchette gauche. C'est une configuration intuitive observée lors des tests, la main droite éjecte toujours le corail (gâchette droite + bouton A) et la main gauche inverse le rouleau (en poussant le corail vers le haut de la rampe). La modification du mappage du contrôleur est l'un des changements de code les plus faciles à effectuer, alors n'hésitez pas à expérimenter et à voir ce que vous aimez ! Pour plus d'informations sur la manière de procéder, voir la section "Modifications" ci-dessous.

6 Apporter des modifications

Cette section détaille certaines modifications courantes que vous pourriez vouloir apporter au code du KitBot et fournit des références sur la manière d'aborder ces modifications.

6.1 Changer les boutons d'action

L'une des modifications les plus faciles à apporter au code d'un robot basé sur les commandes consiste à changer les boutons ou les comportements des boutons qui contrôlent une commande. Les commandes utilisées dans le KitBot 2025 ne se terminent pas (isFinished renvoie toujours false) et ne doivent donc être utilisées qu'avec le comportement whileTrue(), mais il est possible de changer très simplement les boutons auxquels elles correspondent.

Les mappages de boutons dans l'exemple de programme sont effectués vers la fin de la méthode configureBindings() dans le fichier RobotContainer. Les liaisons utilisées pour ce projet sont réalisées à l'aide des méthodes d'aide de la classe CommandXboxController. Ces méthodes d'aide existent pour chaque bouton du contrôleur et renvoient un objet Trigger qui possède d'autres méthodes pouvant être utilisées pour spécifier un comportement pour la liaison.

Tel que fourni, le code relie le bouton **a** à une puissance déterminée. Pour modifier cela, il suffit de remplacer la méthode d'aide a() par la méthode de n'importe quel autre bouton ! Vous pouvez voir toutes les options disponibles en consultant la Javadoc de CommandXboxController pour les méthodes qui ne prennent aucun paramètre et renvoient un objet Trigger.

Par exemple, pour faire passer la commande d'admission du pare-choc a au bouton x, il suffit de remplacer a() par x().

```
// before
operatorController.a()
    .whileTrue(new RollerCommand(() -> RollerConstants.ROLLER_EJECT_VALUE, () -> 0, rollerSubsystem));
// after
operatorController.x()
    .whileTrue(new RollerCommand(() -> RollerConstants.ROLLER_EJECT_VALUE, () -> 0, rollerSubsystem));
```

6.2 Modification du comportement du stick de propulsion

Un autre changement facile à effectuer consiste à modifier quels sticks du contrôleur sont utilisés pour la conduite du robot et de quelle manière. Le code fourni effectue ce mappage lors de la configuration de la commande par défaut du groupe motopropulseur à la fin de configureBindings dans RobotContainer.

L'exemple de code utilise l'axe Y du stick gauche pour avancer et reculer et l'axe X du stick droit pour tourner. Il est facile de les intervertir ou d'en déplacer un pour qu'ils soient sur le seul et même stick ! Pour passer en revue les options disponibles, recherchez les méthodes qui renvoient un nombre flottant **float** dans la documentation API XboxController. Pour effectuer ce type de modification, localisez l'appel de méthode que vous souhaitez modifier, par exemple getLeftY(), et remplacez-le par la nouvelle méthode souhaitée, par exemple getRightY().

Exemple : changer la conduite avant-arrière sur l'axe Y du stick droit et laisser la rotation sur l'axe X droit.

```
53      driveSubsystem.setDefaultCommand(
54          driveSubsystem.driveArcade(
55              driveSubsystem, () -> driverController.getRightY(), () -> driverController.getRightX()));
```

Vous pouvez également modifier les valeurs des axes. Une modification courante consiste porter au cube les valeurs. Cela préserve le signe de la valeur (le positif reste positif, le négatif reste négatif) et la valeur maximale (donc ne réduit pas la vitesse maximale du robot) tout en offrant moins de sensibilité aux faibles entrées, ce qui permet potentiellement un contrôle plus précis à faible vitesse. Pour effectuer ce type de modification, vous pouvez appliquer la modification à l'axe où elle est capturée. La méthode Arcade Drive de la classe Differential Drive porte déjà au carré les entrées par défaut (tout en préservant le signe), vous voudrez probablement désactiver cela si vous les portez au cube vous-même en passant un paramètre supplémentaire à l'appel de la méthode Arcade Drive dans le sous-système de propulsion.

Exemple : modifier uniquement l'axe de rotation pour qu'il soit au cube :

```
driveSubsystem.setDefaultCommand(new DriveCommand(
    () -> -driverController.getLeftY(),
    () -> Math.pow(-driverController.getRightX(),b:3),
    driveSubsystem));
```

```
48  /*Method to control the drivetrain using arcade drive. Arcade drive takes a speed in the X (forward/back) direction
49   * and a rotation about the Z (turning the robot about it's center) and uses these to control the drivetrain motors */
50  public void arcadeDrive(double speed, double rotation) {
51    m_drivetrain.arcadeDrive(speed, rotation, squareInputs:false);
52  }
```

Une autre modification courante consiste à réduire les valeurs par défaut, mais à autoriser la valeur maximale si l'on appuie sur un bouton (mode turbo). Ce type de modification peut également être effectué au moment de la capture, mais lorsque la complexité augmente, vous pouvez souhaiter passer d'une définition de commande en ligne à un autre type où vous pouvez définir le comportement de la commande de manière plus claire.

Exemple : Réduire la conduite avant-arrière de 50 % à moins d'appuyer sur le pare-chocs droit.

```
driveSubsystem.setDefaultCommand(new DriveCommand(
    () -> -driverController.getLeftY() *
      (driverController.getHID().getRightBumperButton() ? 1 : 0.5),
    () -> -driverController.getRightX(),
    driveSubsystem));
```

Cet exemple utilise une construction abrégée "if else" appelée opérateur ternaire. Cet opérateur nous permet d'écrire une simple instruction "if" d'une manière très compacte, si le pare-choc droit est enfoncé, nous transmettons la valeur totale, sinon nous la multiplions par 0,5. Le code utilise également une méthode appelée "getHID()" sur le CommandXboxController ; cette méthode utilise l'objet XboxController sous-jacent que nous utilisons pour obtenir la valeur booléenne directe du bouton au lieu des objets Trigger qui proviennent de la classe CommandXboxController.

6.3 Changement du type de propulsion

Le dernier changement probable que nous aborderons est le passage d'Arcade Drive à Tank Drive. Contrairement à la propulsion Arcade qui associe un axe à la rotation et un autre à la marche avant/arrière, la propulsion Tank associe un axe (généralement l'axe Y) à chaque côté d'une propulsion différentielle. Pour effectuer ce changement, vous devrez aller au-delà du RobotContainer car les sous-systèmes de propulsion fournis n'exposent pas de méthode de propulsion de type Tank. Dans le sous-système de propulsion, créez une nouvelle méthode appelée tankDrive(). Cette méthode devrait ressembler à la méthode arcadeDrive. Modifiez ensuite la correspondance des commandes par défaut dans RobotContainer pour utiliser cette nouvelle méthode avec les axes de joystick appropriés.

Exemple :

```
64    public Command driveTank(
65        CANDriveSubsystem driveSubsystem, DoubleSupplier xSpeed, DoubleSupplier zRotation) {
66      return Commands.run(
67          () -> drive.tankDrive(xSpeed.getAsDouble(), zRotation.getAsDouble()), driveSubsystem);
68    }
```

6.4 Développer des routines autonomes

Le code fourni contient un mode autonome très basique qui avance à 50% de puissance pendant 1 seconde. Des modes autonomes supplémentaires peuvent être développés, soit en ajoutant des méthodes supplémentaires dans le fichier Autos (voir le projet exemple Hatchbot Inlined pour une illustration de ce style avec des autonomes plus complexes), soit en créant des fichiers séparés pour chaque routine autonome (voir le projet Hatchbot Traditional pour cette approche).

Il est courant (mais certainement pas obligatoire !) d'avoir plusieurs routines autonomes que vous pouvez souhaiter exécuter en fonction de différentes positions de départ ou de stratégies. Si vous poursuivez dans cette voie, la façon la plus courante de choisir entre les deux pour chaque match est de le faire à l'aide d'un SendableChooser sur le tableau de bord.