

# BEGINNER JAVA FOR FRC

By: FIRST  
Team 2363  
Triple Helix

# DISCLAIMER

Since the control system is changing we cannot guarantee all of this presentation will be valid for the coming season. However, we have been assured that mostly everything will remain the same.

# AGENDA

- Advantages
- Basic FRC Project Classes(source code files)
- Subsystems
- Commands
- The Scheduler

# ADVANTAGES

- Java is the AP Computer Science language.
- Teaches good Object Oriented design practices.
  - Forces the splitting of code into smaller files.
  - Easier to find different parts of your code.
  - Allows focus on one part of the robot when coding.
  - Makes code easy to test and debug in parts.
  - Makes it easy for multiple people to write multiple parts of the robot at the same time.
- Easy to learn the basics.
- The majority of teams use Java.
- Java is a highly employable language.

# BASIC FRC PROJECT CLASSES

- Subsystem – Defines the parts of a robot.
- Command – Controls the robot subsystems.
- Robot – Controls the running of commands, stores subsystems, and OI.
- OI – Sets up the operator interface devices.
- RobotMap – Sets up the mapping for device channels and other robot constants.

# CREATE ROBOT PROJECT

# SUBSYSTEM

- Any part that needs to act independently of the other parts of the robot.
  - Drivetrain, Shooter, Arm, Elevator, Feeder
- Contains the sensors and motors specific to that robot component and only that robot component.
  - Each motor should only be accessible by one subsystem.
- K.I.S.S
  - Simpler is better.

# SUBSYSTEM CLASS

```
public class ExampleSubsystem extends Subsystem {  
  
    public void initDefaultCommand() {  
        setDefaultCommand(new SomethingCommand());  
    }  
}
```



# SUBSYSTEM METHODS

```
public void initDefaultCommand() {  
    setDefaultCommand(new SomethingCommand());  
}
```

- Sets the command that will run when no other commands are explicitly specified to run.
- A subsystem does not have to have a default command.
- The rest of the methods in the subsystem classes will be written by you.

# DESIGNING A SUBSYSTEM

**Subsystem name**

**Drivetrain**

**Subsystem specific sensors and motors**

**Left Motor  
Right Motor  
Robot Drive Style - tank**

**Subsystem behaviors**

**Drive(forward power, turning power)**

# WRITE DRIVETRAIN SUBSYSTEM

```
public class Drivetrain extends Subsystem {  
    private Talon left = new Talon(1);  
    private Talon right = new Talon(2);  
    private RobotDrive rd = new RobotDrive(left, right);  
    ...  
    public void drive(double left, double right) {  
        rd.tankDrive(left, right);  
    }  
}
```

# THINGS TO REMEMBER - SUBSYSTEMS

- Only make something public if something outside of the subsystem needs it.
  - Make motors and sensors private so only that subsystem has access to them.
  - Make the subsystem behaviors public so the commands can use them to control the robot.
- If two parts of the robot are going to need to be controlled separately, split them into separate subsystems.

# ROBOT CLASS

```
public class Robot extends IterativeRobot {  
    private Command autonomousCommand;  
    public static ExampleSubsystem subsystem = new ExampleSubsystem();  
    public static OI oi = new OI();  
  
    public void robotInit() {  
        autonomousCommand = new ExampleCommand();  
    }  
  
    public void autonomousInit() {  
        autonomousCommand.start();  
    }  
  
    public void autonomousPeriodic() {  
        Scheduler.getInstance().run();  
    }  
  
    public void teleopInit() {  
        autonomousCommand.cancel();  
    }  
  
    public void teleopPeriodic() {  
        Scheduler.getInstance().run();  
    }  
}
```

# ROBOT METHODS

```
public void robotInit() {  
    autonomousCommand = new ExampleCommand();  
}
```

- This is executed when the robot is turned on.
- This is a good place to set the autonomous command that you would like to run.

# ROBOT METHODS

```
public void autonomousInit() {  
    autonomousCommand.start();  
}
```

- This is called at the beginning of the autonomous period.
- This is also a good place to select an autonomous command.
- This is where you start the autonomous command that you selected.

# ROBOT METHODS

```
public void autonomousPeriodic() {  
    Scheduler.getInstance().run();  
}
```

- This is repeated over and over again throughout the autonomous period.
- The Scheduler controls what commands are executing at any given time. Do not remove this line of code!



# ROBOT METHODS

```
public void teleopInit() {  
    autonomousCommand.cancel();  
}  
  
public void teleopPeriodic() {  
    Scheduler.getInstance().run();  
}
```

- This is basically the same as the autonomous period methods but for the teleoperated period instead.
- The teleopInit() method is where you need to cancel the autonomous command if it is still running.

# DECLARE THE DRIVETRAIN

```
public class Robot extends IterativeRobot {  
  
    public static Drivetrain drivetrain = new Drivetrain();  
  
    ...  
}
```

# COMMANDS

- Commands control the subsystems using the methods defined in each subsystem.
- Commands will execute repeatedly until they either finish or something interrupts them.
- A command can control more than one subsystem at a time, but only one command can be running on a subsystem at any given time.
- Default commands can be defined for subsystems that will run when no other commands are specified to run for that subsystem.

# COMMAND CLASS

```
public class ExampleCommand extends Command {  
    public ExampleCommand() {  
        requires(Robot.drivetrain);  
    }  
  
    protected void initialize() {  
    }  
  
    protected void execute() {  
    }  
  
    protected boolean isFinished() {  
        return false;  
    }  
  
    protected void end() {  
    }  
  
    protected void interrupted() {  
    }  
}
```

# COMMAND METHODS

```
public ExampleCommand() {  
    requires(Robot.drivetrain);  
}
```

- Defines which subsystem(s) the command is going to use.
- Multiple subsystems can be required by a single command.
- Only one command can require a subsystem at a time.

# COMMAND METHODS

```
protected void initialize() {
```

```
}
```

- Anything that would need to happen before the command starts.
- Example: Getting the starting position off of an encoder or potentiometer.

# COMMAND METHODS

```
protected void execute() {
```

```
}
```

- The action that needs to be repeated over and over again.
- Example: Reading the values from a joystick and sending them to the drivetrain.

# COMMAND METHODS

```
protected boolean isFinished() {  
    return false;  
}
```

- This method is checked after the execute method runs and will stop the command if the method returns true.
- Default commands should return false.
- Commands that only run once should return true.



# COMMAND METHODS

```
protected void end() {
```

```
}
```

- This method will run after the `isFinished()` method returns a true value.
- This should contain anything that needs to happen at the end of a command.
- Example: Set the drivetrain motors to stop.

# COMMAND METHODS

```
protected void interrupted() {
```

```
}
```

- This command will run if another command takes control of a subsystem that this command was using.
- This should contain anything that needs to happen if the command is interrupted.
- Example: Setting a motor to stop.

# WRITE DRIVE COMMAND

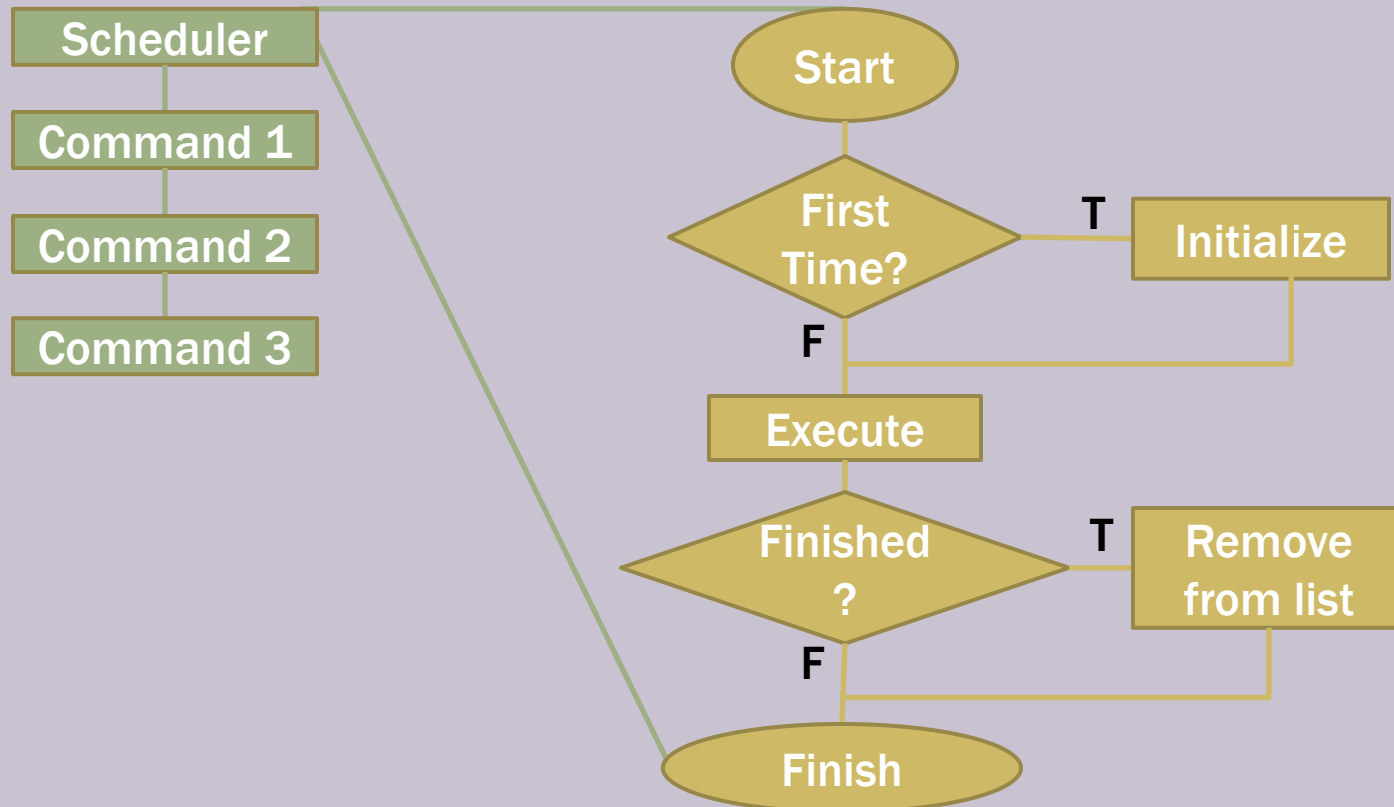
```
public class DriveCommand extends Command {  
    public DriveCommand() {  
        requires(Robot.drivetrain);  
    }  
    ...  
    public void execute() {  
        Robot.drivetrain.drive(0.5, 0.5);  
    }  
    ...  
}
```

# THINGS TO REMEMBER - COMMANDS

- Make sure to use `requires(subsystem)`, this will help avoid issues with multiple commands trying to control the same subsystem.
- Commands are not multi-threaded, if a command takes a long time to finish one execution the robot will lock up.
- Default commands should return false for the `isFinished()` method.
- If you want a command to only run once, return true in `isFinished()` immediately (Firing a pneumatic cylinder).
- Default commands need to be set in the subsystem, don't forget to go back and set them.

# THE SCHEDULER

- Keeps a list of the currently running commands.
- Executes all commands one at a time every ~20 ms.



# 01

- This is where you will declare any of your operator controls (joysticks).
- This is also where you can map different buttons to activate different commands.

# OI CLASS

```
public class OI {  
    public Joystick stick = new Joystick(port);  
  
    public OI () {  
        JoystickButton button = new JoystickButton(stick, 1);  
  
        button.whenPressed(new ExampleCommand());  
        button.whileHeld(new ExampleCommand());  
        button.whenReleased(new ExampleCommand());  
    }  
}
```

# MAPPING BUTTONS

- Create the button
  - `JoystickButton button;`
- Set which joystick and button it will use.
  - `button = new JoystickButton(stick, 1);`
- Set what command will fire when the button is used.
  - `button.whenPressed(new ExampleCommand())`
    - Will execute when the button is first pressed down.
  - `button.whenReleased(new ExampleCommand())`
    - Will execute when the button is first released.
  - `button.whileHeld(new ExampleCommand())`
    - Will repeatedly execute while the button is held down. The `interrupted()` method of the command will be executed when the button is released.



# WRITE STOP COMMAND

```
public class StopCommand extends Command {  
    public DriveCommand() {  
        requires(Robot.drivetrain);  
    }  
    ...  
    public void execute() {  
        Robot.drivetrain.drive(o, 0);  
    }  
    ...  
}
```

# WRITE OI

```
public class OI {  
    public Joystick leftJoystick;  
    public Joystick rightJoystick;  
  
    public OI() {  
        leftJoystick = new Joystick(1);  
        rightJoystick = new Joystick(2);  
  
        JoystickButton stopDrivingButton = new JoystickButton(leftJoystick, 1);  
        stopDrivingButton.whileHeld(new StopCommand());  
    }  
}
```

# WRITE JOYSTICK DRIVE COMMAND

```
public class JoystickDriveCommand extends Command {  
  
    public JoystickDriveCommand() {  
        requires(Robot.drivetrain);  
    }  
  
    ...  
  
    protected void execute() {  
        Robot.drivetrain.drive(Robot.oi.joystick.getY(), Robot.oi.joystick.getY());  
    }  
  
    ...  
}
```

# SET JOYSTICK DRIVE AS DEFAULT

```
public class Drivetrain extends Subsystem {  
    ...  
    public void initDefaultCommand() {  
        setDefaultCommand(new JoystickDriveCommand());  
    }  
    ...  
}
```

# ROBOTMAP CLASS

```
public class RobotMap {  
    public static final int leftMotor = 1;  
    public static final int rightMotor = 2;  
  
    public static final int rangefinderPort = 1;  
    public static final int rangefinderModule = 1;  
  
    public static final double shooterPower = 0.75;  
}
```

# ROBOTMAP

- This is where you can set up all of your device channel configurations.
- This allows you to have all of your configuration information in one place.
- This is also a good place to set up constant values to use in your code (shooter power).

# IN REVIEW

- Design and write the subsystem.
- Add it to the Robot class.
- Write any commands for the subsystem.
- Add the default command to the subsystem if there is one.
- Map any OI controls to the commands.
- Add any autonomous functionality to autonomous commands.
- Enjoy your robot!

# SUGGESTIONS

- Use lots of comments.
  - `//something something something`
  - Use these to describe what certain code does.
- Descriptive variable names.
- Make sure to check your spelling.
- Use robot map and constants, it will help you quickly change values later.
- Small subsystems are better.
- Use buttons and very basic commands to test basic functionality of your robot.
- Save often, like every time you type a line of code often.



# QUESTIONS?

- Contact me at [lythgoe.matthew@gmail.com](mailto:lythgoe.matthew@gmail.com).