

INTERMEDIATE 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

- Command Groups
- Reusable Commands
- Pneumatics
- Sensors
- Questions

COMMAND GROUPS

A command group is a command that executes a list of commands in sequence, parallel, or both.

They can be used to create more complex commands out of very basic commands.

COMMAND GROUPS - SEQUENTIAL

addSequential(Command command)

addSequential(Command command, double timeout)

Adds a command to the list, the next command will not start until the added command is finished.

COMMAND GROUPS - PARALLEL

`addParallel(Command command)`

`addParallel(Command command, double timeout)`

Adds a command to the list, the next command will start right away.

COMMAND GROUPS - EXAMPLE

```
public class AutonomousCommand extends CommandGroup {  
  
    public AutonomousCommand() {  
        addParallel(new DeployArm());  
        addSequential(new DriveForward(), 3);  
        addSequential(new DriveStop());  
    }  
}
```

COMMAND GROUPS – WAIT COMMAND

The wait command can be used to add pauses or durations into a command group.

```
addSequential(new WaitCommand(5));
```


COMMAND GROUPS – WAIT COMMAND EXAMPLE

```
public class AutonomousCommand extends CommandGroup {  
  
    public AutonomousCommand() {  
        addParallel(new DeployArm());  
        addParallel(new Drive(0.75), 3);  
        addSequential(new WaitCommand(3));  
        addSequential(new Drive(0));  
    }  
}
```

REUSABLE COMMANDS

It can be beneficial to create one reusable command instead of several separate commands.

Separate Commands	Reusable Command
new Drive25Percent()	new Drive(0.25)
new Drive50Percent()	new Drive(0.5)
new Drive75Percent()	new Drive(0.75)
new Drive100Percent()	new Drive(1)
new DriveStop()	new Drive(0)

REUSABLE COMMANDS - CONSTRUCTOR

Use the constructor of a command to make the command configurable.

The constructor is the method that has the same name as the class, it is called when you create an instance of the class.

```
public Drive(double power) {  
    ...  
}
```

REUSABLE COMMANDS - FIELD

After the constructor is called the value passed in will need stored somewhere.

Fields (instance variables) can be used to store these values. Fields are a global variable that can be used in any of the methods.

```
private double power;
```

REUSABLE COMMANDS - EXAMPLE

```
public class Drive extends Command {  
  
    private double powerField;  
  
    public Drive(double power) {  
        powerField = power;  
        requires(drivetrain);  
    }  
}
```

...

REUSABLE COMMANDS - EXAMPLE

...

```
public void execute() {  
    drivetrain.drive(powerField);  
}
```

...

```
}
```

PNEUMATICS

A Pneumatic system in code consists of two things:

- Compressor
- Solenoids

PNEUMATICS - COMPRESSOR

Compressor takes care of itself, just needs configured with the correct ports and started.

Start the compressor in the RobotTemplate class in the robotInit() method.

```
new Compressor(1, 1).start();  
(pressure switch DIO port, relay port)
```


PNEUMATICS - SOLENOIDS

Solenoids are used to fire pneumatic cylinders in one direction or the other. Unlike motors, Solenoids only need set once.

Single acting cylinders need one Solenoid.

Double acting cylinders need two Solenoids, or one DoubleSolenoid.

PNEUMATICS - SOLENOIDS

Solenoid pickup1 = new Solenoid(1);

Solenoid pickup2 = new Solenoid(2);

(solenoid module port)

**DoubleSolenoid pickupDouble = new
DoubleSolenoid(3, 4);**

(solenoid module port 1, solenoid module port 2)

PNEUMATICS - ACTUATION

To actuate a solenoid use the `set()` method.

To actuate a single acting cylinder:

```
pickup1.set(true);
```

To deactuate a single acting cylinder:

```
puickup1.set(false);
```

PNEUMATICS - ACTUATION

To actuate a double acting cylinder using separate Solenoids:

```
pickup1.set(true);  
pickup2.set(false);
```

To deactuate a double acting cylinder using separate Solenoids:

```
pickup1.set(false);  
pickup2.set(true);
```

PNEUMATICS - ACTUATION

To actuate a double acting cylinder using a DoubleSolenoid:

```
pickupDouble.set(DoubleSolenoid.value.kForward);
```

To deactuate a double acting cylinder using a DoubleSolenoid:

```
pickupDouble.set(DoubleSolenoid.value.kReverse);
```

PNEUMATICS - EXAMPLE

```
public class Pickup extends Subsystem {  
  
    private Solenoid solenoid1 = new Solenoid(1);  
    private Solenoid solenoid2 = new Solenoid(2);  
  
    public boolean deployArm(boolean deploy) {  
        solenoid1.set(deploy);  
        solenoid2.set(!deploy);  
    }  
}
```

REUSABLE COMMAND – EXAMPLE 2

Reusable command for the pickup subsystem.

Pickup states:

- Deployed and On
- Deployed and Off
- Retracted and On
- Retracted and Off (Default state)

REUSABLE COMMAND – EXAMPLE 2

```
public class PickupCommand extends Command {  
  
    private boolean deployArm;  
    private double rollerPower;  
  
    public PickupCommand(boolean deployArm, double rollerPower) {  
        this.deployArm = deployArm;  
        this.rollerPower = rollerPower;  
        requires(pickup);  
    }  
  
    public void initialize() {  
        pickup.deployArm(deployArm);  
    }  
  
    ...  
}
```


REUSABLE COMMAND – EXAMPLE 2

...

```
public void execute() {  
    pickup.setRoller(rollerPower);  
}
```

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

...

SENSORS

- Limit Switches – Touch sensor.
- Encoders – Distance or speed.
- Gyros – Direction or turning rate.

LIMIT SWITCHES

Limit switches can be used to determine if a manipulator is in a certain position, i.e. the end of its range of motion.

They are declared using the `DigitalInput` class.

```
DigitalInput upperElevatorLimit = new DigitalInput(2);  
(DI port the limit switch is plugged into)
```

LIMIT SWITCHES - EXAMPLE

```
public class Elevator extends Subsystem {  
  
    private DigitalInput elevatorUpperLimit = new DigitalInput(2);  
    private DigitalInput elevatorLowerLimit = new DigitalInput(3);  
  
    public boolean isUpperLimitReached() {  
        return elevatorUpperLimit.get();  
    }  
  
    public boolean isLowerLimitReached() {  
        return elevatorLowerLimit.get();  
    }  
  
    ...  
}
```

ENCODERS

Encoders can be used to determine how fast something is rotating/moving or to determine how far something has rotated/moved.

Encoders are declared using the Encoder class.

```
Encoder armEncoder = new Encoder(5, 6);
```

(port channel A is plugged into, port channel B is plugged into)

ENCODERS - DATA

Encoders have several different values you can get from them.

getRate()

Returns the rate the encoder is currently turning at in distance per second.

getStopped()

Returns true if the encoder is stopped.

getDirection()

Returns the direction the encoder last turned in.

getDistance()

Returns the distance the encoder has rotated since starting.

ENCODERS - CONFIGURATION

- Encoders have a couple of settings that can be configured to get more meaningful values from it.

setDistancePerPulse(double d)

This method will set the distance each pulse is scaled to. This will change the values returned by `getDistance()` and `getRate()` to give you more meaningful values.

setMinRate(double min)

This method will set the minimum rate that the encoder is considered to be moving. This will effect when the `getStopped()` method returns true.

ENCODERS – START/STOP

An encoder doesn't start reading values until it is told to start reading. Likewise it does not stop reading values until it is told to stop.

start()

Starts the reading of values.

stop()

Stops the reading of values.

reset()

Resets the encoders count to 0.

ENCODERS - EXAMPLE

```
public class Drivetrain extends Subsystem {  
    private Encoder left = new Encoder(5, 6);  
  
    public Drivetrain() {  
        left.setDistancePerPulse(0.004);  
        left.setMinRate(0.5);  
        left.start();  
    }  
  
    public double getDistanceTravelled() {  
        return left.getDistance();  
    }  
}
```

GYROS

Gyros can be used to get the robots heading relative to its starting position or the rate that the robot is turning.

Gyros are declared using the Gyro class.

```
Gyro gyro = new Gyro(1);  
(analog port the gyro is plugged into)
```

GYROS - DATA

Gyros have two values that they report.

getAngle()

Returns the angle in degrees the robot is currently facing relative to the original facing.

getRate()

Returns the rate in degrees per second that the robot is currently turning.

GYROS - CONFIGURATION

There is only one configuration that needs set for gyros, this value can be found in the data for the specific gyro being used.

setSensitivity(double voltsPerDegreePerSecond)

This sets the sensitivity level of the gyro, if this value is not set correctly the angle the gyro reports will drift over time.

GYROS - EXAMPLE

```
public class Drivetrain extends Subsystem {  
  
    private Gyro gyro = new Gyro(1);  
  
    public Drivetrain() {  
        gyro.setSensitivity(0.6);  
    }  
  
    public double getCurrentFacing() {  
        return gyro.getAngle();  
    }  
}
```

QUESTIONS?

- Contact me at lythgoe.matthew@gmail.com.