

# Creating Simple Commands

This article describes the basic format of a Command and walks through an example of creating a command to drive your robot with Joysticks.

## Basic Command Format

To implement a command, a number of methods are overridden from the WPILib Command class. Most of the methods are boiler plate and can often be ignored, but are there for maximum flexibility when you need it. There a number of parts to this basic command class:

```
C++

#include "MyCommandName.h"

/*
    * 1.      Constructor - Might have parameters for this command such as target
positions of devices. Should also set the name of the command for debugging purposes.
    * This will be used if the status is viewed in the dashboard. And the command
should require (reserve) any devices is might use.
    */
MyCommandName::MyCommandName() : CommandBase("MyCommandName")
{
    Requires(Elevator);
}

// initialize() - This method sets up the command and is called immediately before the
command is executed for the first time and
// every subsequent time it is started . Any initialization code should be here.
void MyCommandName::Initialize()
{
}

/*
    *      execute() - This method is called periodically (about every 20ms) and does the
work of the command. Sometimes, if there is a position a
    * subsystem is moving to, the command might set the target position for the subsystem in
```

# Creating Simple Commands

```
initialize() and have an empty execute() method.
*/
void MyCommandName::Execute()
{

}

bool MyCommandName::IsFinished()
{
    return false;
}

void MyCommandName::End()
{

}

// Make this return true when this Command no longer needs to run execute()
void MyCommandName::Interrupted()
{

}
```

## Java

```
public class MyCommandName extends Command {

    /*
     * 1.      Constructor - Might have parameters for this command such as target
positions of devices. Should also set the name of the command for debugging purposes.
     * This will be used if the status is viewed in the dashboard. And the command
should require (reserve) any devices is might use.
     */
    public MyCommandName() {
        super("MyCommandName");
        requires(elevator);
    }
}
```

# Creating Simple Commands

```
//      initialize() - This method sets up the command and is called immediately
before the command is executed for the first time and every subsequent time it is started .
// Any initialization code should be here.
protected void initialize() {
}

/*
 *      execute() - This method is called periodically (about every 20ms) and does
the work of the command. Sometimes, if there is a position a
 *      subsystem is moving to, the command might set the target position for the
subsystem in initialize() and have an empty execute() method.
 */
protected void execute() {
}

// Make this return true when this Command no longer needs to run execute()
protected boolean isFinished() {
    return false;
}
}
```

## Simple Command Example

This example illustrates a simple command that will drive the robot using tank drive with values provided by the joysticks.

**C++**

```
#include "DriveWithJoysticks.h"
#include "RobotMap.h"

DriveWithJoysticks::DriveWithJoysticks() : CommandBase("DriveWithJoysticks")
{
    Requires(Robot::drivetrain); // Drivetrain is our instance of the drive system
}
```

# Creating Simple Commands

```
// Called just before this Command runs the first time
void DriveWithJoysticks::Initialize()
{

}

/*
 * execute() - In our execute method we call a tankDrive method we have created in our
 subsystem. This method takes two speeds as a parameter which we get from methods in the OI
 class.
 * These methods abstract the joystick objects so that if we want to change how we get
 the speed later we can do so without modifying our commands
 * (for example, if we want the joysticks to be less sensitive, we can multiply them
 by .5 in the getLeftSpeed method and leave our command the same).
 */
void DriveWithJoysticks::Execute()
{
    Robot::drivetrain->Drive(Robot::oi->GetJoystick());
}

/*
 * isFinished - Our isFinished method always returns false meaning this command never
 completes on it's own. The reason we do this is that this command will be set as the
 default command for the subsystem. This means that whenever the subsystem is not running
 another command, it will run this command. If any other command is scheduled it will
 interrupt this command, then return to this command when the other command completes.
 */
bool DriveWithJoysticks::IsFinished()
{
    return false;
}

void DriveWithJoysticks::End()
{
    Robot::drivetrain->Drive(0, 0);
}

// Called when another command which requires one or more of the same
// subsystems is scheduled to run
```

# Creating Simple Commands

```
void DriveWithJoysticks::Interrupted()
{
    End();
}
```

## Java

```
public class DriveWithJoysticks extends Command {

    public DriveWithJoysticks() {
        requires(drivetrain); // drivetrain is an instance of our Drivetrain subsystem
    }

    protected void initialize() {
    }

    /*
     * execute() - In our execute method we call a tankDrive method we have created in our
     subsystem. This method takes two speeds as a parameter which we get from methods in the OI
     class.
     * These methods abstract the joystick objects so that if we want to change how we get
     the speed later we can do so without modifying our commands
     * (for example, if we want the joysticks to be less sensitive, we can multiply them
     by .5 in the getLeftSpeed method and leave our command the same).
     */
    protected void execute() {
        drivetrain.tankDrive(oi.getLeftSpeed(), oi.getRightSpeed());
    }

    /*
     * isFinished - Our isFinished method always returns false meaning this command never
     completes on it's own. The reason we do this is that this command will be set as the
     default command for the subsystem. This means that whenever the subsystem is not running
     another command, it will run this command. If any other command is scheduled it will
     interrupt this command, then return to this command when the other command completes.
     */
    protected boolean isFinished() {
```

# Creating Simple Commands

```
        return false;
    }

    protected void end() {
    }

    protected void interrupted() {
    }
}
```