

# WPILIB PROGRAMMING

# WPILib programming

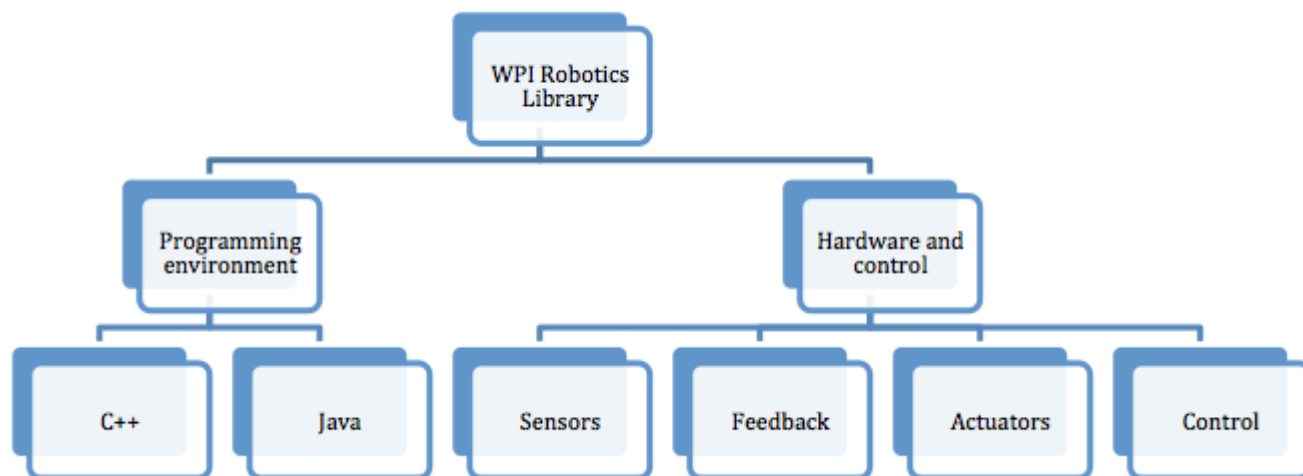
Basic WPILib Programming features .....	3
What is WPILib.....	3
Operating the robot with feedback from sensors (PID control) .....	6
Getting your robot to drive with the RobotDrive class .....	10
WPILib sensors .....	15
Gyros to control robot driving direction .....	15
Robot to driver station networking .....	18
Writing a simple NetworkTables program in C++ and Java with a Java client (PC side).....	18
Using TableViewer to see NetworkTable values.....	28

# Basic WPILib Programming features

## What is WPILib

The WPI Robotics library (WPILib) is a set of software classes that interfaces with the hardware and software in your FRC robot's control system. There are classes to handle sensors, motor speed controllers, the driver station, and a number of other utility functions such as timing and field management. In addition, WPILib supports many commonly used sensors that are not in the kit, such as ultrasonic rangefinders.

## What's included in the library



There are three versions of the library, one for each supported language. This document specifically deals with the text-based languages, C++ and Java. There is considerable effort to keep the APIs for Java and C++ very similar with class names and method names being the same. There are some differences that reflect language differences such as pointers vs. references, name case conventions, and some minor differences in functionality. These languages were chosen because they represent a good level of abstraction for robot programs than previously used languages. The WPI Robotics Library is designed for maximum extensibility and software reuse with these languages.

WPILib has a generalized set of features, such as general-purpose counters, to provide support for custom hardware and devices. The FPGA hardware also allows for interrupt processing to be dispatched at the task level, instead of as kernel interrupt handlers, reducing the complexity of many common real-time issues.

Fundamentally, C++ offers the highest performance possible for your robot programs. Java on the other hand has acceptable performance and includes extensive run-time checking of your program to make it much easier to debug and detect errors. Those with extensive programming experience can probably make their own choices, and beginning might do better with Java to take advantage of the ease of use.

There is a detailed list of the differences between C++ and Java on [Wikipedia available here](#). Below is a summary of the differences that will most likely effect robot programs created with WPILib.

## Java programming with WPILib

```

91 public void autonomousInit() {
92     isAuto = true;
93     CommandBase.shooter.zeroRPMOffsets();
94     CommandBase.turret.zeroAngleOffsets();
95     // instantiate the command used for the autonomous period
96     autonomousCommand = (Command) (OI.getInstance().auton.getSelected());
97     // schedule the autonomous command (example)
98     autonomousCommand.start();
99 }

```

- Java objects must be allocated manually, but are freed automatically when no references remain.
- References to objects instead of pointers are used. All objects must be allocated with the new operator and are referenced using the dot (.) operator (e.g. gyro.getAngle()).
- Header files are not necessary and references are automatically resolved as the program is built.
- Only single inheritance is supported, but interfaces are added to Java to get most of the benefits that multiple inheritance provides.
- Checks for array subscripts out of bounds, uninitialized references to objects and other runtime errors that might occur in program development.
- Compiles to byte code for a virtual machine, and must be interpreted.

## C++ programming with WPILib

```

void Claw::Open() {
    victor->Set(1);
}

void Claw::Close() {
    victor->Set(-1);
}

void Claw::Stop() {
    victor->Set(0);
}

```

- Memory allocated and freed manually.
- Pointers, references, and local instances of objects.
- Header files and preprocessor used for including declarations in necessary parts of the program.
- Implements multiple inheritance where a class can be derived from several other classes, combining the behavior of all the base classes.
- Does not natively check for many common runtime errors.
- Highest performance on the platform, because it compiles directly to machine code for the PowerPC processor in the cRIO.

## Operating the robot with feedback from sensors (PID control)

Without feedback the robot is limited to using timing to determine if it's gone far enough, turned enough, or is going fast enough. And for mechanisms, without feedback it's almost impossible to get arms at the right angle, elevators at the right height, or shooters to the right speed. There are a number of ways of getting these mechanisms to operate in a predictable way. The most common is using PID (Proportional, Integral, and Differential) control. The basic idea is that you have a sensor like a potentiometer or encoder that can measure the variable you're trying to control with a motor. In the case of an arm you might want to control the angle - so you use a potentiometer to measure the angle. The potentiometer is an analog device, it returns a voltage that is proportional to the shaft angle of the arm.

To move the arm to a preset position, say for scoring, you predetermine what the potentiometer voltage should be at that preset point, then read the arms current angle (voltage). The difference between the current value and the desired value represents how far the arm needs to move and is called the error. The idea is to run the motor in a direction that reduces the error, either clockwise or counterclockwise. And the amount of error (distance from your setpoint) determines how fast the arm should move. As it gets closer to the setpoint, it slows down and finally stops moving when the error is near zero.

The WPILib `PIDController` class is designed to accept the sensor values and output motor values. Then given a setpoint, it generates a motor speed that is appropriate for its calculated error value.

## Creating a PIDController object

```
Joystick turretStick(1);
Jaguar turretMotor(1);
AnalogChannel turretPot(1);
PIDController turretControl(0.1, 0.001, 0.0, &turretPot, &turretMotor);

turretControl.Enable(); // start calculating PIDOutput values

while(IsOperator())
{
    turretControl.SetSetpoint((turretStick.GetX() + 1.0) * 2.5); 1
    Wait(.02);           // wait for new joystick values
}
```

The **PIDController** class allows for a PID control loop to be created easily, and runs the control loop in a separate thread at consistent intervals. The **PIDController** automatically checks a **PIDSource** for feedback and writes to a **PIDOutput** every loop. Sensors suitable for use with **PIDController** in WPILib are already subclasses of **PIDSource**. Additional sensors and custom feedback methods are supported through creating new subclasses of **PIDSource**. Jaguars and Victors are already configured as subclasses of **PIDOutput**, and custom outputs may also be created by sub-classing **PIDOutput**.

A potentiometer that turns with the turret will provide feedback of the turret angle. The potentiometer is connected to an analog input and will return values ranging from 0-5V from full clockwise to full counterclockwise motion of the turret. The joystick X-axis returns values from -1.0 to 1.0 for full left to full right. We need to scale the joystick values to match the 0-5V values from the potentiometer. This is done with the expression (1). The scaled value can then be used to change the setpoint of the control loop as the joystick is moved.

The 0.1, 0.001, and 0.0 values are the Proportional, Integral, and Differential coefficients respectively. The **AnalogChannel** object is already a subclass of **PIDSource** and returns the voltage as the control value and the Jaguar object is a subclass of **PIDOutput**.

The **PIDController** object will automatically (in the background):

- Read the **PIDSource** object (in this case the turretPot analog input)
- Compute the new result value
- Set the **PIDOutput** object (in this case the turretMotor)

This will be repeated periodically in the background by the **PIDController**. The default repeat rate is 50ms although this can be changed by adding a parameter with the time to the end of the **PIDController** argument list. See the reference document for details.

## Setting the P, I, and D values

The output value is computed by adding the weighted values of the error (proportional term), the sum of the errors (integral term) and the rate of change of errors (differential term). Each of these is multiplied by a scaling constant, the P, I and D values before adding the terms together. The constants allow the PID controller to be tuned so that each term is contributing an appropriate value to the final output.

The P, I, and D values are set in the constructor for the PIDController object as parameters.

The [SmartDashboard](#) in Test mode has support for helping you tune PID controllers by displaying a form where you can enter new P, I, and D constants and test the mechanism.

## Continuous sensors like continuous rotation potentiometers

The PIDController object can also handle continuous rotation potentiometers as input devices. When the pot turns through the end of the range the values go from 5V to 0V instantly. The PID controller method SetContinuous() will set the PID controller to a mode where it will compute the shortest distance to the desired value which might be through the 5V to 0V transition. This is very useful for drive trains that use have continuously rotating swerve wheels where moving from 359 degrees to 10 degrees should only be a 11 degree motion, not 349 degrees in the opposite direction.

## Controlling the speed of a motor

Controlling motor speed is a little different than position control. Remember, with position control you are setting the motor value to something related to the error. As the error goes to zero the motor stops running. If the sensor (an optical encoder for example) is measuring motor speed as the speed reaches the setpoint, the error goes to zero, and the motor slows down. This causes the motor to oscillate as it constantly turns on and off. What is needed is a base value of motor speed called the "Feed forward" term. This 4th value, F, is added in to the output motor voltage independently of the P, I, and D calculations and is a base speed the motor will run at. The P, I, and D values adjust the feed forward term (base motor speed) rather than directly control it. The closer the feed forward term is, the smoother the motor will operate.

**Note:** The feedforward term is multiplied by the setpoint for the PID controller so that it scales with the desired output speed.



## Using PID controllers in command based robot programs

```

1 package org.usfirst.frc190.GearsBot.subsystems;
2 import edu.wpi.first.wpilibj.*;
3 import edu.wpi.first.wpilibj.command.PIDSubsystem;
4 import edu.wpi.first.wpilibj.livewindow.LiveWindow;
5 import org.usfirst.frc190.GearsBot.RobotMap;
6
7 public class Elevator extends PIDSubsystem {
8     SpeedController motor = RobotMap.elevatorMotor;
9     AnalogChannel pot = RobotMap.elevatorPot;
10
11     public Elevator() {
12         super("Elevator", 1.0, 0.0, 0.0);
13         setAbsoluteTolerance(0.2);
14         getPIDController().setContinuous(false);
15         LiveWindow.addActuator("Elevator", "PIDSubsystem Controller", getPIDController());
16     }
17
18     public void initDefaultCommand() {
19     }
20
21     protected double returnPIDInput() {
22         return pot.getAverageVoltage();
23     }
24
25     protected void usePIDOutput(double output) {
26         motor.pidWrite(output);
27     }
28 }
29

```

The easiest way to use PID controllers with command based robot programs is by implementing PIDSubsystems for all your robot mechanisms. This is simply a subsystem with a PIDController object built-in and provides a number of convenience methods to access the required PID parameters. In a command based program, typically commands would provide the setpoint for different operations, like moving an elevator to the low, medium or high position. In this case, the `isFinished()` method of the command would check to see if the embedded PIDController had reached the target. See the Command based programming section for more information and examples.

# Getting your robot to drive with the RobotDrive class

WPILib provides a RobotDrive object that handles most cases of driving the robot either in autonomous or teleop modes. It is created with either two or four speed controller objects. There are methods to drive with either Tank, Arcade, or Mecanum modes either programmatically or directly from Joysticks.

*Note: the examples illustrated in this section are generally correct but have not all been tested on actual robots. But should serve as a starting point for your projects.*

## Creating a RobotDrive object with Jaguar speed controllers

```
RobotDrive drive(1, 2, 3, 4); // four motor drive configuration
```

```
RobotDrive drive(1, 2); // left, right motors on ports 1,2
```

Create the RobotDrive object specifying either two or four motor ports. By default the RobotDrive constructor will create Jaguar class instances attached to each of those ports.

## Using other types of speed controllers

```

0
7 public class RobotTemplate extends SimpleRobot {
8
9     RobotDrive myDrive;
10    Victor frontLeft, frontRight, rearLeft, rearRight;
11
12    public void robotInit() {
13        frontLeft = new Victor(1);
14        frontRight = new Victor(2);
15        rearLeft = new Victor(3);
16        rearRight = new Victor(4);
17        myDrive = new RobotDrive(frontLeft, rearLeft, frontRight, rearRight);
18    }
19
20    public void autonomous() {
21
22    }

```

You can use RobotDrive with other types of speed controllers as well. In this case you must create the speed controller objects manually and pass the references or pointers to the RobotDrive constructor.

These are Java programs but the C++ program is very similar.

## Tank driving with two joysticks

```

8  public class RobotTemplate extends SimpleRobot {
9
10     RobotDrive myDrive;
11     Joystick left, right;
12
13     public void robotInit() {
14         myDrive = new RobotDrive(1, 2, 3, 4);
15         left = new Joystick(1);
16         right = new Joystick(2);
17     }
18
19     public void autonomous() {
20     }
21
22     public void operatorControl() {
23         while (isOperatorControl() && isEnabled()) {
24             myDrive.tankDrive(left, right);
25             Timer.delay(0.01);
26         }
27     }
28

```

In this example a RobotDrive object is created with 4 default Jaguar speed controllers. In the operatorControl method the RobotDrive instance tankDrive method is called and it will select the Y-axis of each of the joysticks by default. There are other versions of the tankDrive method that can be used to use alternate axis or just numeric values.

## Arcade driving with a single joystick

```

8  public class RobotTemplate extends SimpleRobot {
9
10     RobotDrive myDrive;
11     Joystick driveStick;
12
13     public void robotInit() {
14         myDrive = new RobotDrive(1, 2, 3, 4);
15         driveStick = new Joystick(1);
16     }
17
18     public void autonomous() {
19     }
20
21     public void operatorControl() {
22         while (isOperatorControl() && isEnabled()) {
23             myDrive.arcadeDrive(driveStick);
24             Timer.delay(0.01);
25         }
26     }
27

```

Similar to the example above a single joystick can be used to do single-joystick driving (called arcade). In this case, the X-axis is selected by default for the turn axis and the Y-axis is selected for the speed axis.

## Autonomous driving using the RobotDrive object

```

8
9  public class RobotTemplate extends SimpleRobot {
10
11      RobotDrive myDrive;
12      Joystick driveStick;
13      Gyro gyro;
14      static final double Kp = 0.03;
15
16      public void robotInit() {
17          myDrive = new RobotDrive(1, 2, 3, 4);
18          gyro = new Gyro(1);
19      }
20
21      public void autonomous() {
22          while (isAutonomous() && isEnabled()) {
23              double angle = gyro.getAngle();
24              myDrive.arcadeDrive(-1.0, -angle * Kp);
25              Timer.delay(0.01);
26          }
27      }
28

```

The RobotDrive object also has a number of features that makes it ideally suited for autonomous control. This example illustrates using a gyro for driving in a straight line (the current heading) using the arcade method for steering. While the robot continues to drive in a straight line the gyro headings are roughly zero. As the robot veers off in one direction or the other, the gyro headings vary either positive or negative. This is very convenient since the arcade method turn parameter is also either positive or negative. The magnitude of the gyro headings sets the rate of the turn, that is more off zero the gyro heading is, the faster the robot should turn to correct.

This is a perfect use of proportional control where the rate of turn is proportional to the gyro heading (being off zero). The heading is in values of degrees and can easily get pretty far off, possibly as much as 10 degrees as the robot drives. But the values for the turn in the arcade method are from zero to one. To solve this problem, the heading is scaled by a constant to get it in the range required by the turn parameter of the arcade method. This parameter is called the proportional gain, often written as  $K_P$ .

In this particular example the robot is designed such that negative speed values go forward. Also, not that the the angle from the gyro is written as "-angle". This is because this particular robot turns in the opposite direction from the gyro corrections and has to be negated to correct that. Your robot might be different in both cases depending on gearing and motor connections.

## Mecanum driving

```

8
9 public class RobotTemplate extends SimpleRobot {
10
11     RobotDrive myDrive;
12     Joystick moveStick, rotateStick;
13
14     public void robotInit() {
15         myDrive = new RobotDrive(1, 2, 3, 4);
16         moveStick = new Joystick(1);
17         rotateStick = new Joystick(2);
18     }
19
20     public void autonomous() {
21     }
22
23     public void operatorControl() {
24         while (isOperatorControl() && isEnabled()) {
25             myDrive.mecanumDrive_Polar(moveStick.getY(), moveStick.getY(), rotateStick.getTwist());
26             Timer.delay(0.01);
27         }
28     }
29

```

The RobotDrive can also handle Mecanum driving. That is using Mecanum wheels on the chassis to enable the robot to drive in any direction without first turning. This is sometimes called Holonomic driving.

In this example there are two joysticks controlling the robot. moveStick supplies the direction vector for the robot, that is which way it should move irrespective of the heading. rotateStick supplies the rate of rotation in the twist (rudder) axis on the joystick. If you push the moveStick full forward the robot will move forward, even if it's facing to the left. At the same time, if you rotate the rotateStick, the robot will spin in the rotation direction with the rotation rate from the amount of twist, while the robot continues to move forward.

# WPILib sensors

## Gyros to control robot driving direction

Gyros typically in the FIRST kit of parts are provided by Analog Devices, and are actually angular rate sensors. The output voltage is proportional to the rate of rotation of the axis perpendicular to the top package surface of the gyro chip. The value is expressed in mV/°/second (degrees/second or rotation expressed as a voltage). By integrating (summing) the rate output over time, the system can derive the relative heading of the robot.

Another important specification for the gyro is its full-scale range. Gyros with high full-scale ranges can measure fast rotation without “pinning” the output. The scale is much larger so faster rotation rates can be read, but there is less resolution due to a much larger range of values spread over the same number of bits of digital to analog input. In selecting a gyro, you would ideally pick the one that had a full-scale range that matched the fastest rate of rotation your robot would experience. This would yield the highest accuracy possible, provided the robot never exceeded that range.

### Using the Gyro class

The Gyro object should be created in the constructor of the **RobotBase** derived object. When the Gyro object is used, it will go through a calibration period to measure the offset of the rate output while the robot is at rest to minimize drift. This requires that the robot be stationary and the gyro is unusable until the calibration is complete.

Once initialized, the **GetAngle()** (or **getAngle()** in Java) method of the Gyro object will return the number of degrees of rotation (heading) as a positive or negative number relative to the robot's position during the calibration period. The zero heading can be reset at any time by calling the **Reset()** (**reset()** in Java) method on the Gyro object.

See the code samples below for an idea of how to use the Gyro objects.

### Setting Gyro sensitivity

The Gyro class defaults to the settings required for the 80°/sec gyro that was delivered by FIRST in the 2008 kit of parts. It is important to check the documentation included with the kit to ensure that you have the correct sensitivity setting.

To change gyro types call the **SetSensitivity(float sensitivity)** method (or **setSensitivity(double sensitivity)** in Java) and pass it the sensitivity in volts/°/sec. Take note that the units are typically specified in mV (volts / 1000) in the spec sheets. For example, a sensitivity of 12.5 mV/°/sec would require a **SetSensitivity()** (**setSensitivity()** in Java) parameter value of 0.0125.

## Using a gyro to drive straight

The following example programs cause the robot to drive in a straight line using the gyro sensor in combination with the **RobotDrive** class. The **RobotDrive.Drive** method takes the speed and the turn rate as arguments; where both vary from -1.0 to 1.0. The gyro returns a value indicating the number of degrees positive or negative the robot deviated from its initial heading. As long as the robot continues to go straight, the heading will be zero. This example uses the gyro to keep the robot on course by modifying the turn parameter of the Drive method.

The angle is multiplied by a proportional scaling constant ( $K_p$ ) to scale it for the speed of the robot drive. This factor is called the proportional constant or loop gain. Increasing  $K_p$  will cause the robot to correct more quickly (but too high and it will oscillate). Decreasing the value will cause the robot correct more slowly (possibly never reaching the desired heading). This is known as proportional control, and is discussed further in the PID control section of the advanced programming section.

```
class GyroSample : public SimpleRobot
{
    RobotDrive myRobot; // robot drive system
    Gyro gyro;
    static const float Kp = 0.03;

public:
    GyroSample():
        myRobot(1, 2),    // initialize the sensors in initialization list
        gyro(1)
    {
        myRobot.SetExpiration(0.1);
    }

    void Autonomous()
    {
        gyro.Reset();
        while (IsAutonomous())
        {
            float angle = gyro.GetAngle(); // get heading
            myRobot.Drive(-1.0, -angle * Kp); // turn to correct heading
            Wait(0.004);
        }
        myRobot.Drive(0.0, 0.0); // stop robot
    }
};
```



## Sample Java program for driving straight

This is a sample Java program that drives in a straight line. See the comments in the C++ example (previous step) for an explanation of its operation.

```
package edu.wpi.first.wpilibj.templates;

import edu.wpi.first.wpilibj.Gyro;
import edu.wpi.first.wpilibj.RobotDrive;
import edu.wpi.first.wpilibj.SimpleRobot;
import edu.wpi.first.wpilibj.Timer;

public class GyroSample extends SimpleRobot {

    private RobotDrive myRobot; // robot drive system
    private Gyro gyro;

    double Kp = 0.03;

    public GyroSample()
    {
        myRobot.setExpiration(0.1);
    }

    protected void Autonomous() {
        gyro.reset();
        while (isAutonomous()) {
            double angle = gyro.getAngle(); // get heading
            myRobot.drive(-1.0, -angle*Kp); // drive to heading
            Timer.delay(0.004);
        }
        myRobot.drive(0.0, 0.0); // stop robot
    }
};
```

# Robot to driver station networking

## Writing a simple NetworkTables program in C++ and Java with a Java client (PC side)

NetworkTables is an implementation of a distributed "dictionary". That is named values are created either on the robot, driver station, or potentially an attached coprocessor, and the values are automatically distributed to all the other participants. For example, a driver station laptop might receive camera images over the network, perform some vision processing algorithm, and come up with some values to sent back to the robot. The values might be an X, Y, and Distance. By writing these results to NetworkTable values called "X", "Y", and "Distance" they can be read by the robot shortly after being written. Then the robot can act upon them.

NetworkTables can be used by programs on the robot in either C++, Java or LabVIEW and is built into each version of WPILib.

## Using NetworkTables from a Java robot program

```

1  package edu.wpi.first.wpilibj.templates;
2
3  import edu.wpi.first.wpilibj.SimpleRobot;
4  import edu.wpi.first.wpilibj.Timer;
5  import edu.wpi.first.wpilibj.networktables.NetworkTable;
6
7  public class EasyNetworkTableExample extends SimpleRobot {
8
9      NetworkTable table; 1
10
11  public void robotInit() {
12      table = NetworkTable.getTable("datatable"); 2
13  }
14
15  public void autonomous() {
16  }
17
18  public void operatorControl() {
19      double x = 0;
20      double y = 0;
21      while (isOperatorControl() && isEnabled()) {
22          Timer.delay(0.25);
23          table.putNumber("X", x); 3
24          table.putNumber("Y", y);
25          x += 0.05;
26          y += 1.0;
27      }
28  }
29  }
30

```

NetworkTables programs on the robot are easiest to write. The program simply reads or writes values from within the program. The instance of NetworkTables is automatically created by the WPILib runtime system. This example is the simplest robot program that can be written that continuously writes pairs of values (X, and Y) to a table called "datatable". Whenever these values are written on the robot, they can be read shortly after on the desktop client.

1. The variable "table" is of type NetworkTable. NetworkTables are hierarchical, that is tables can be nested by using their names for representing the position in the hierarchy.
2. The table is associated with values within the hierarchy, in this case the path to the data is /datatable/X and /datatable/Y.
3. Values are written to the "datatable" NetworkTable. Each value will automatically be replicated between all the NetworkTable programs running on the network.

When this program is run on the robot and enabled in Teleop mode, it will start writing incrementing X and Y values continuously, updating them 4 times per second (every 0.25 seconds).

## Using Network Tables from a C++ robot program



```

#include "WPILib.h"
#include "NetworkTables/NetworkTable.h"

class RobotDemo : public SimpleRobot
{
public:
    NetworkTable *table; 1

    RobotDemo(void) {
        table = NetworkTable::GetTable("datatable");
    } 2

    void OperatorControl(void) {
        double x = 0;
        double y = 0;
        while (IsOperatorControl() && IsEnabled()) {
            Wait(1.0);
            table->PutNumber("X", x); 3
            table->PutNumber("Y", y);
            x += 0.25;
            y += 0.25;
        }
    }
};

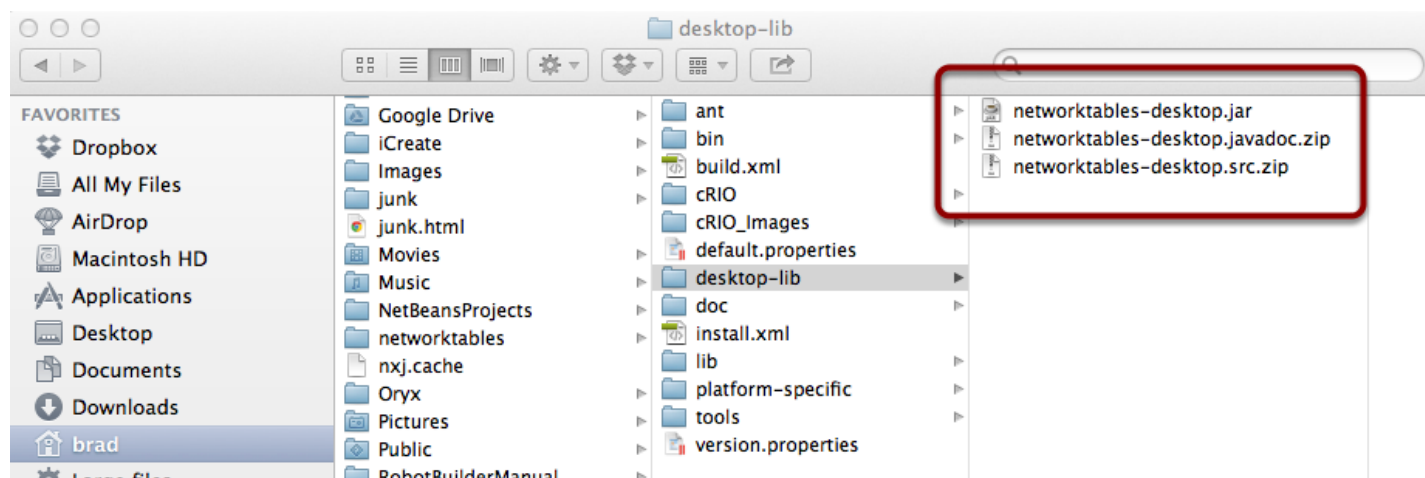
START_ROBOT_CLASS(RobotDemo);
  
```

NetworkTables programs on the robot are easiest to write. The program simply reads or writes values from within the program. The instance of NetworkTables is automatically created by the WPILib runtime system. This example is the simplest robot program that can be written that continuously writes pairs of values (X, and Y) to a table called "datatable". Whenever these values are written on the robot, they can be read shortly after on the desktop client.

1. The variable "table" is of type NetworkTable. NetworkTables are hierarchical, that is tables can be nested by using their names for representing the position in the hierarchy.
2. The table is associated with values within the hierarchy, in this case the path to the data is /datatable/X and /datatable/Y.
3. Values are written to the "datatable" NetworkTable. Each value will automatically be replicated between all the NetworkTable programs running on the network.

When this program is run on the robot and enabled in Teleop mode, it will start writing incrementing X and Y values continuously, updating them 4 times per second (every 0.25 seconds).

## Using the client version of NetworkTables on a desktop computer



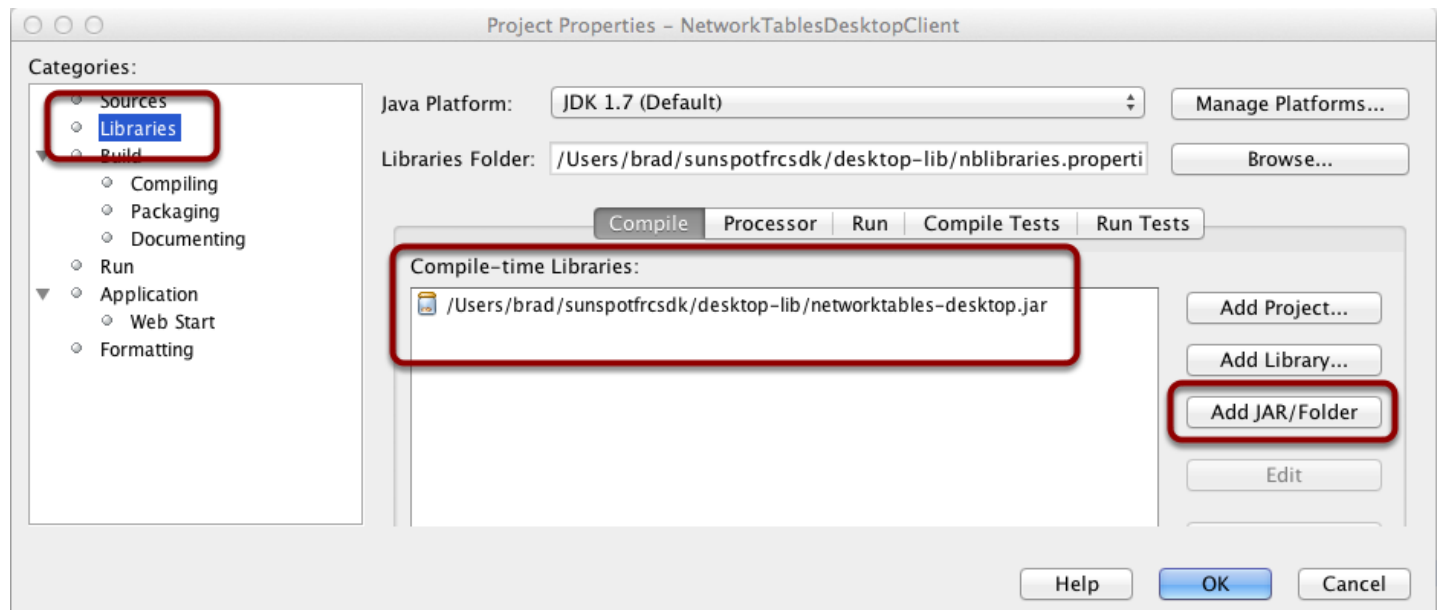
The NetworkTables libraries are built into all versions of robot-side WPILib. You can set values from the robot in C++, Java or LabVIEW with simple put and get methods. To use it on a laptop (usually the driver station computer), there are several options:

1. a client library that you can reference from Java programs that you write.
2. from plugins that you write for the SmartDashboard (it's included there)

The Java library is part of the NetBeans Java plugin installation and can be found in the <user-directory>/sunspotfrcsdk/desktop-lib directory as shown here.

**For C++ WindRiver installations the .jar files are located in the C:\WindRiver\WPILib\desktop-lib directory.**

## Setting up NetBeans to create the client-side (laptop/desktop computer) program



To write a program that runs on your PC that uses NetworkTables the Java project must reference the JAR file from the NetBeans installation shown above. The project has to reference the networktables-desktop.jar file. This is an example of doing it with NetBeans but any IDE will have a way of adding .JAR files to a project. In this example the .jar file was added to the project properties.

**Note:** this is not necessary for a robot program since NetworkTables is built into WPILib. You simply have to add the necessary java import statements or C++ #includes for the NetworkTable classes that are used in the program.

## The client (laptop) side of the program

```

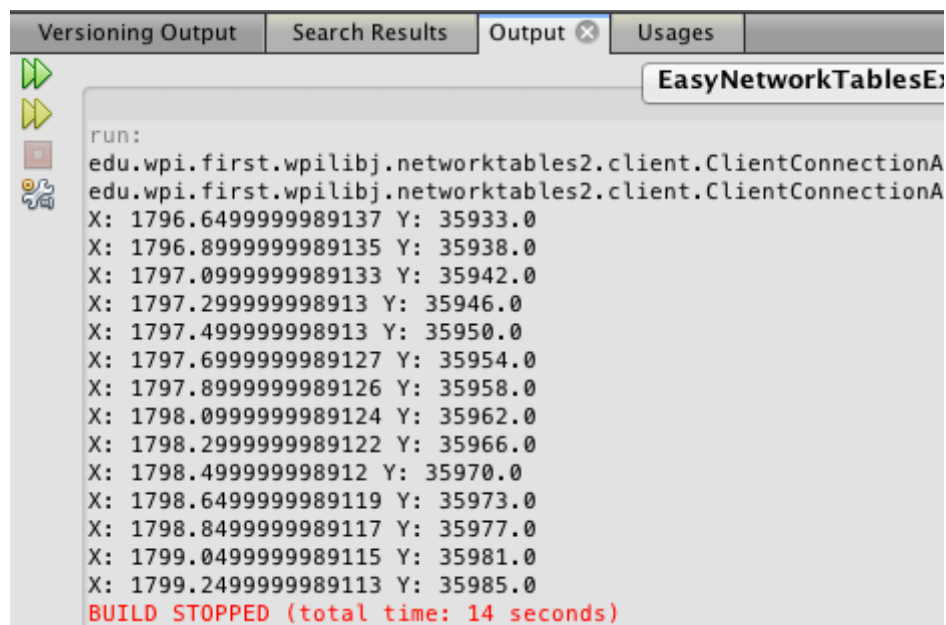
1  package networktablesdesktopclient;
2
3  import edu.wpi.first.wpilibj.networktables.NetworkTable;
4  import java.util.logging.Level;
5  import java.util.logging.Logger;
6
7  public class NetworkTablesDesktopClient {
8
9      public static void main(String[] args) {
10         new NetworkTablesDesktopClient().run();
11     }
12
13     public void run() {
14
15         NetworkTable.setClientMode();           1
16         NetworkTable.setIPAddress("10.1.90.2");
17         NetworkTable table = NetworkTable.getTable("datatable");  2
18
19         while (true) {
20             try {
21                 Thread.sleep(1000);
22             } catch (InterruptedException ex) {    3
23                 Logger.getLogger(NetworkTablesDesktopClient.class.getName()).log(Level.SEVERE, null, ex);
24             }
25
26             double x = table.getNumber("X", 0.0);  4
27             double y = table.getNumber("Y", 0.0);
28             System.out.println("X: " + x + " Y: " + y);
29         }
30     }
31 }
32

```

This program is the simplest program that you can write on a PC to use NetworkTables. It continuously reads the values from robot example in the previous step.

1. Set NetworkTables to client mode (not on the robot) and specify the IP address of the robot.
2. Create a NetworkTable variable ("table") that is associated with the "datatable" NetworkTable.
3. Loop continuously and sleep for 1 second each time through the loop.
4. Read the X and Y values from the /datatable NetworkTable that was written on the robot in the previous program and print the values. The program output is shown below.

## Program output from the simple client example



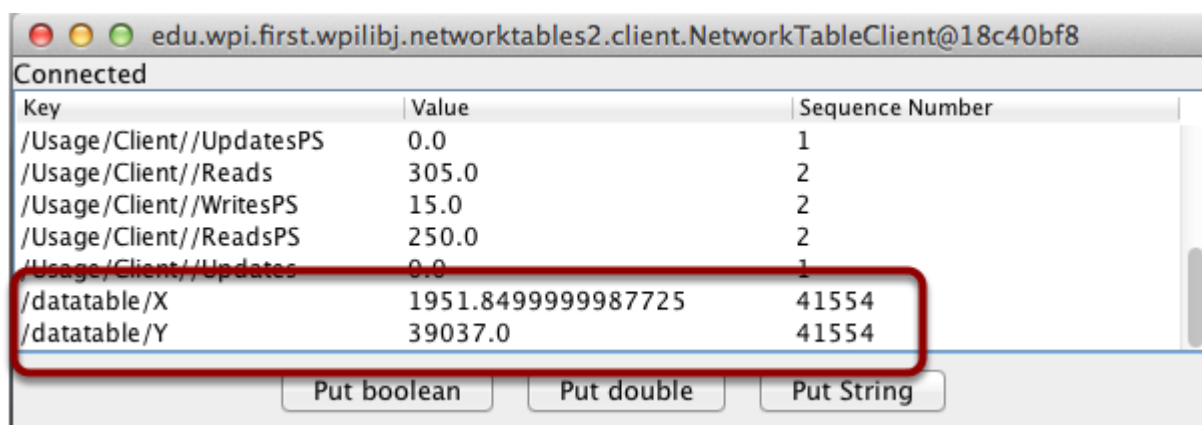
```

run:
edu.wpi.first.wpilibj.networktables2.client.ClientConnectionA
edu.wpi.first.wpilibj.networktables2.client.ClientConnectionA
X: 1796.6499999989137 Y: 35933.0
X: 1796.8999999989135 Y: 35938.0
X: 1797.0999999989133 Y: 35942.0
X: 1797.299999998913 Y: 35946.0
X: 1797.499999998913 Y: 35950.0
X: 1797.6999999989127 Y: 35954.0
X: 1797.8999999989126 Y: 35958.0
X: 1798.0999999989124 Y: 35962.0
X: 1798.2999999989122 Y: 35966.0
X: 1798.499999998912 Y: 35970.0
X: 1798.6499999989119 Y: 35973.0
X: 1798.8499999989117 Y: 35977.0
X: 1799.0499999989115 Y: 35981.0
X: 1799.2499999989113 Y: 35985.0
BUILD STOPPED (total time: 14 seconds)

```

This output is from the NetBeans "output" window. This is the results from the `System.out.println()` method from the previous program that is running on a desktop computer retrieving values written on the robot from the earlier Robot program.

## Viewing the NetworkTables variables in TableViewer



Key	Value	Sequence Number
/Usage/Client//UpdatesPS	0.0	1
/Usage/Client//Reads	305.0	2
/Usage/Client//WritesPS	15.0	2
/Usage/Client//ReadsPS	250.0	2
/Usage/Client//Updates	0.0	1
/datatable/X	1951.8499999987725	41554
/datatable/Y	39037.0	41554

There is a diagnostic tool called TableViewer that will display the current state of the NetworkTables table. In this case, running it will show the current values of all the variables in the variables created in this example are shown in the red box above. TableViewer is located in the sunspotfrcsdk folder for NetBeans installs or in the C:\WindRiver\Workbench\WPILib folder for C++ installs.



## Receiving notifications of changes to a NetworkTable

```

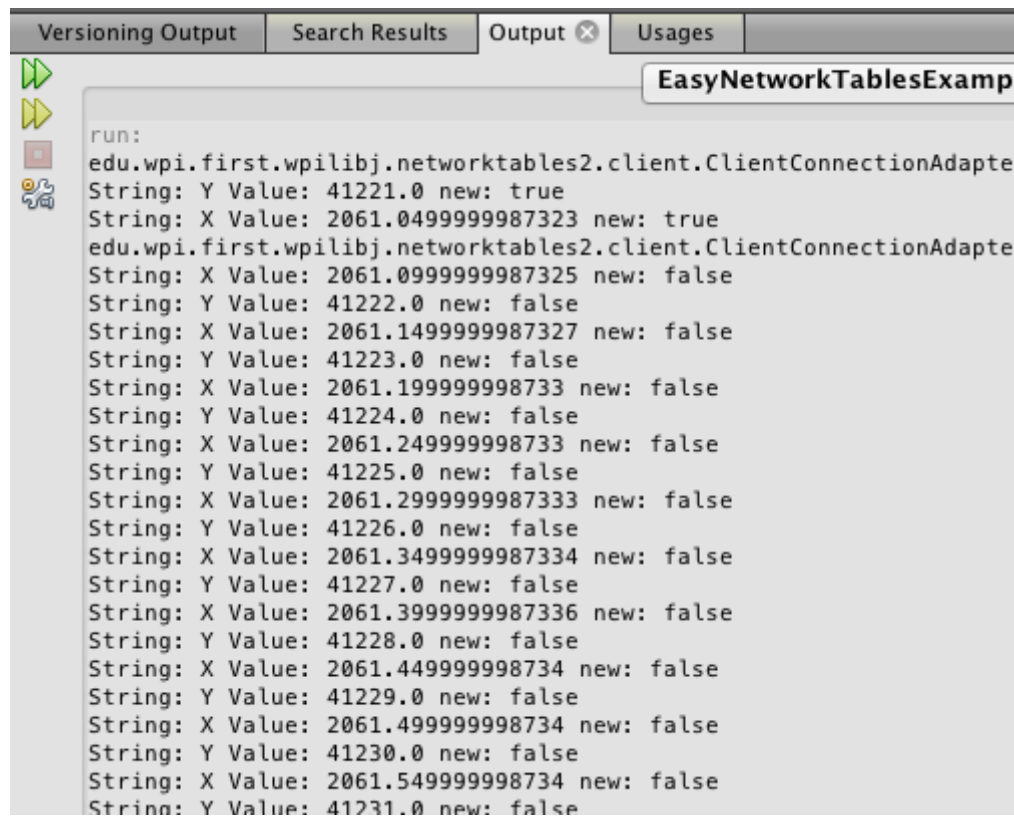
1  package networktablesdesktopclient;
2
3  import edu.wpi.first.wpilibj.networktables.NetworkTable;
4  import edu.wpi.first.wpilibj.tables.ITable;
5  import edu.wpi.first.wpilibj.tables.ITableListener;
6  import java.util.logging.Level;
7  import java.util.logging.Logger;
8
9  public class TableListenerExample implements ITableListener {
10
11     public static void main(String[] args) {
12         new TableListenerExample().run();
13     }
14
15     public void run() {
16         NetworkTable.setClientMode(); 1
17         NetworkTable.setIPAddress("10.1.90.2");
18         NetworkTable table = NetworkTable.getTable("datatable");
19
20         table.addTableListener(this); 2
21
22         try {
23             Thread.sleep(100000);
24         } catch (InterruptedException ex) { 3
25             Logger.getLogger(TableListenerExample.class.getName()).log(Level.SEVERE, null, ex);
26         }
27     }
28
29     @Override
30     public void valueChanged(ITable itable, String string, Object o, boolean bln) { 4
31         System.out.println("String: " + string + " Value: " + o + " new: " + bln);
32     }
33 }
34

```

A PC or Robot program can receive notifications of changes to a NetworkTable. This example is a client-side (PC) program, but the same concepts will work on a robot program. These notifications are received asynchronously as the new values are received by the NetworkTable library.

1. Connect to the NetworkTable server using the same technique as in the previous example.
2. Register this class as a ITableListener. Changes to the "datatable" will be reported to this class through the "valueChanged" callback method (below)
3. Sleep for 100 seconds while values are reported. The program could do anything here, but in this simple example, it only waits for 100 seconds while waiting for values to arrive.
4. This valueChanged method is called whenever there are changes or additions to the NetworkTable "datatable". The boolean value bln will be true if this is a new value or false if it is just an update to a previously reported variable. The Object is the new value that has been received. The output from this program is shown in the next step.

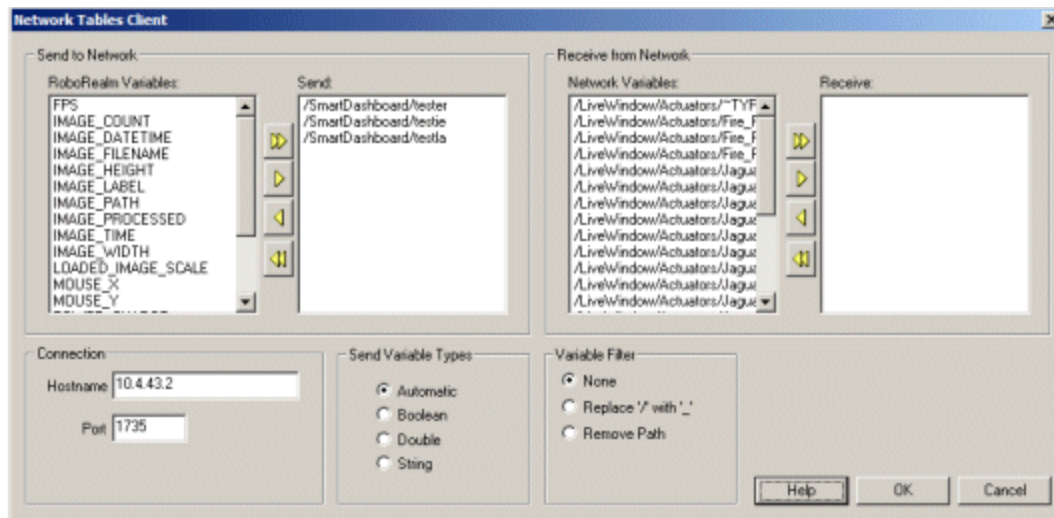
## Results of running the client-side (PC) TableListener example



```
run:
edu.wpi.first.wpilibj.networktables2.client.ClientConnectionAdapte
String: Y Value: 41221.0 new: true
String: X Value: 2061.0499999987323 new: true
edu.wpi.first.wpilibj.networktables2.client.ClientConnectionAdapte
String: X Value: 2061.0999999987325 new: false
String: Y Value: 41222.0 new: false
String: X Value: 2061.1499999987327 new: false
String: Y Value: 41223.0 new: false
String: X Value: 2061.199999998733 new: false
String: Y Value: 41224.0 new: false
String: X Value: 2061.249999998733 new: false
String: Y Value: 41225.0 new: false
String: X Value: 2061.2999999987333 new: false
String: Y Value: 41226.0 new: false
String: X Value: 2061.3499999987334 new: false
String: Y Value: 41227.0 new: false
String: X Value: 2061.3999999987336 new: false
String: Y Value: 41228.0 new: false
String: X Value: 2061.449999998734 new: false
String: Y Value: 41229.0 new: false
String: X Value: 2061.499999998734 new: false
String: Y Value: 41230.0 new: false
String: X Value: 2061.549999998734 new: false
String: Y Value: 41231.0 new: false
```

In this screen image the values returned from the TableListener example are shown. Notice that at the top of the output X and Y are returned with their respective values and "true" for the boolean value. This indicates that they are new values. In all the other cases, the boolean value is "false" indicating that it is just an update to a previously reported value.

## Using NetworkTables with RoboRealm



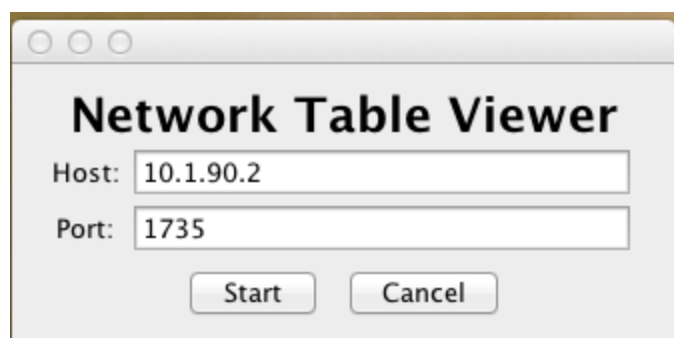
RoboRealm is a program that does client-side (PC) vision processing. RoboRealm can connect to a camera on a robot and do real-time tracking of field targets and sending the results back to the robot. In the past this required writing custom networking code for the PC to robot communications. RoboRealm now has a built-in NetworkTables client and this allows the RoboRealm program to send values directly back to the robot via some shared variables.

For further information see: [http://www.roborealm.com/help/Network\\_Tables.php](http://www.roborealm.com/help/Network_Tables.php)

# Using TableViewer to see NetworkTable values

TableViewer is a program to help debug NetworkTables applications. It acts as a NetworkTables client and allows the viewing of all the keys and associated values in a tabular format. You can use this to quickly see the value of a variable or set a value for a variable. This is a java program making it platform independent - it can run anywhere that the java runtime is installed.

## Starting TableViewer



TableViewer is a java application and is distributed as a .jar file. It is named with the version number, so the actual name you'll see will be dependent on the version of the build. It should be located in the tools directory in either the C++ or Java installation. In the case of C++ it will be in C:\WindRiver\WPILib and in Java it will be in *<user-home-directory>/sunspotfrcsdk/tools*, where the *<user-home-directory>* is the operating system installed users home directory. On some operating systems this can be started by simply double-clicking on the TableViewer.jar file using a file browser. On other systems it might have to be explicitly run from a command line by entering, "java -jar TableViewer.jar". The TableViewer application

Once it is running, enter the host IP address of the robot. This is the FRC standard IP naming convention, 10.TE.AM.2 where TE.AM are replaced with the team number. For example it would be 10.1.90.2 for Team 190.

## Viewing Table Values



The TableView will start up and show all the keys (variable names) and values for those keys. In addition it shows a sequence number which is an internal NetworkTables field used to determine if values are updated and need refreshing. The sequence number increments every time the value of a NetworkTable variable changes. The values wrap around at 65535.

The table rows can be sorted by either the Key, Value or Sequence Number by clicking on the column heading in the table.