WEATHER STATION RECEIVER LAB MANUAL

Table of Contents

WEEK 1 4

Objectives for this week: 4

Arduino UNO Overview. 4

TASK 1 Download the Arduino IDE (Integrated Development Environment). 4

TASK 2 Unpack your team kit and remove the Arduino UNO. 5

COM (Communications) Port 5

Arduino blinking L.E.D. programs 6

TASK 3 Writing, compiling and uploading your BLINK program. 6

TASK 4 Writing, compiling and uploading a Morse code program 8

Fritzing Overview 10

TASK 1. Installing Fritzing 10

Weather Shield: Physical and Electrical Overview 14

Reading Assignment for next week. 16

WEEK 2 17

Objectives for this week: 17

Weather Sensors and how they communicate. 17

Weather Shield schematic 18

Weather Shield Schematic diagram 18

The purpose of Libraries in software coding 18

Example Firmware 19

WEEK 3 33

Objectives for this week: 33

The coding part 35

WEEK 4 41

Objectives for this week: 41

WEEK 5 73

Objectives for this week: 73

WEEK 6 74
Objectives for this week: 74
WEEK 7 75
Objectives for this week: 75
MakerBot 75
Adding GPS information to our weather shield 78
WEEK 8 86
Objectives for this week: 86
WEEK 9 87
Objective for this week: 87
Week 11 88
Working with the Arduino Due 88
Due vs. Uno 88
Your first Arduino Due program 89
Week 12 91
Infrared Light Sensor 91
Week 13 97
Carbon Monoxide Sensor 97
Week 14 100
Rain Gauge 100

WEEK 1

Objectives for this week:

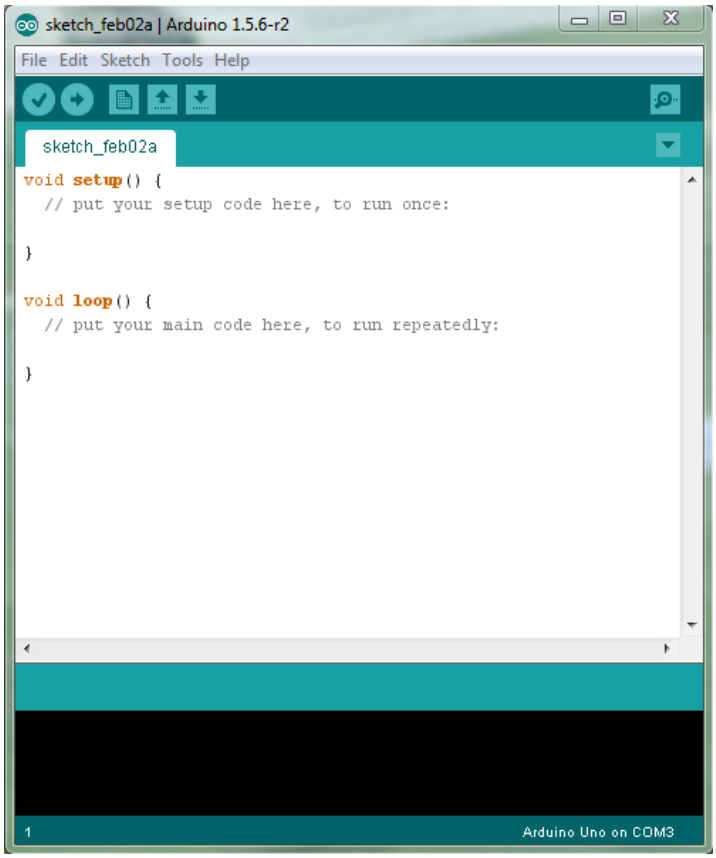
- Introduce ourselves and what we expect from this workshop
- Introduce the workshop process
- What you will need to work at home or in your own workshop
- Introduce the overall weather station system design
- Introduce the Arduino Uno board and embedded microcontroller
- Introduce Arduino shields
- Introduce Fritzing software
- Introduction to electronic device bus / wire communication types
- Introduction to computer programming languages
- Download the Arduino IDE
- Understanding COM ports
- Identify and assemble parts to demo your 'BLINK' script
- Write your first Arduino 'BLINK' script
- Modify your BLINK script
- Create a Fritzing diagram of the project to date.

Arduino UNO Overview.

TASK 1 Download the Arduino IDE (Integrated Development Environment).

We will use the IDE download from http://arduino.cc/en/Main/Software to develop our code, compile it, debug it and upload it to the Arduino board. The code will then run on the Arduino processor and will output any results to the IDE monitor on a COM port as selected.

- Go to the Arduino site and create an account on the site so that you may access additional information and services.
- Download the IDE noting the version of IDE you are downloading. This may take a while depending on your internet speed. Install the *.exe file.

- 

Arduino IDE from http://arduino.cc/en/Main/Software

TASK 2 Unpack your team kit and remove the Arduino UNO.

- We will watch the video https://www.youtube.com/watch?v=5F054MNB1QI and note the pin connections and component types shown in the video and the UNO we have to work with.
- Remove the components in your team box. You will need both the Arduino board and USB A-B cable that connects to your PC. (This is the same cable as used by the Lego Robot projects).
- Handle the board by the edges as static electricity can damage the electronics on the board.

COM (Communications) Port

You will need to identify the COM port your Arduino is communicating on.

Serial ports have been in use for many years and the Operating system's device driver will assign one. The name "serial" is because it takes a byte of data and transmits the 8 bits

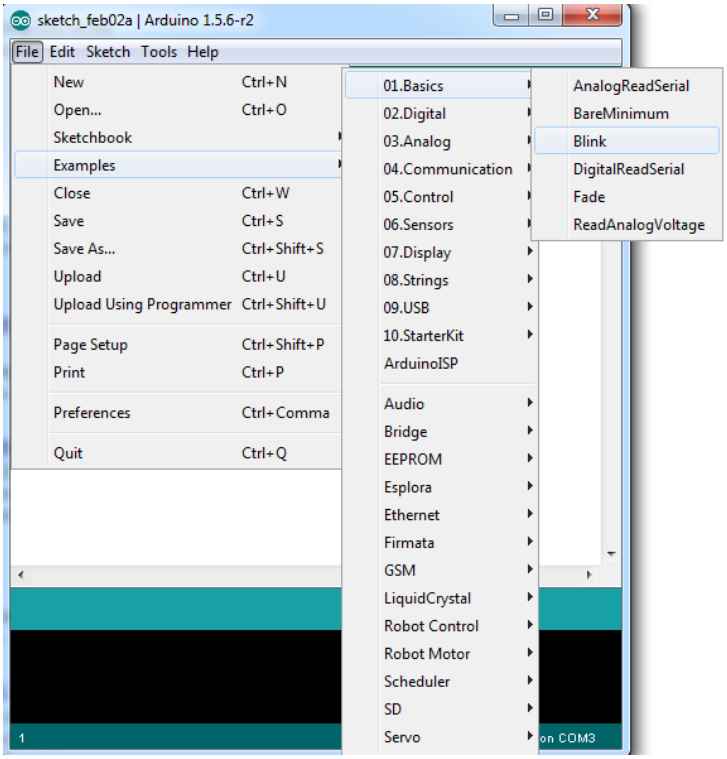
of the byte one at a time; in a series. Serial ports, also called COM ports, are bi-directional which allows your PC and the Arduino to receive data as well as transmit it.

Serial ports use a controller chip, the Universal Asynchronous Receiver/Transmitter (UART). The UART chip takes the parallel output of the computer's system bus and transforms it into serial form for transmission through the serial port. The most recent UART, the 16550, has a 16-byte buffer that can get filled before the computer's processor needs to handle the data. While most standard serial ports have a maximum transfer rate of 115 Kbps (kilobits per second), high speed serial ports can reach data transfer rates of 460 Kbps.

Arduino blinking L.E.D. programs

TASK 3 Writing, compiling and uploading your BLINK program.

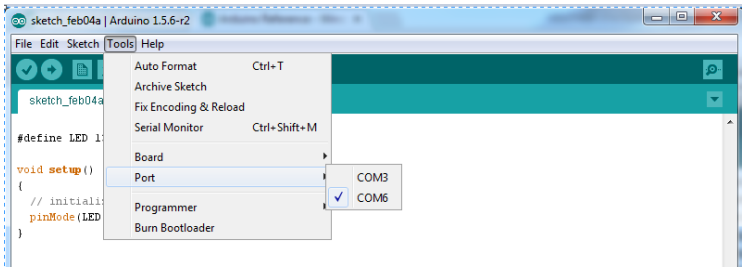
- Watch the following video https://www.youtube.com/watch?v=dnPPoetX0uw in order to correctly place the L.E.D. in your kit.
- The L.E.D. has a polarity to watch for; the short lead with flat lens detent is the negative pin and goes into the Ground location on the header of the UNO.
- Mouse over to File > Examples > 01. Basics > Blink and click on BLINK to open the sketch.



File > Examples > 01. Basics > Blink

There are a few more settings you will need to check in order for a successful upload to happen … this url will give you such guidance
file:///C:/Program%20Files%20(x86)/Arduino/reference/Guide_Windows.html

- List the settings in the Table below which you need to check such the COM port which is assigned during the device driver installation by the Operating System:



Setting	Path	Value
Board	Tools > Board	
COM port	Tools > Serial Port	

Settings Table:

If your upload is unsuccessful, use Google to search your error code.

Congratulations you will now see the LED blinking.

- Now change the timing of the LED blink by modifying the BLINK program code and explain below how and where you will so this.

```
1. void loop() {
2. digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
3. delay(1000);                 // wait for a second
4. digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
5. delay(1000);                 // wait for a second
```

- **New code snippet (replace the xxxx's with _____):**

```
1. void loop() {
2. digitalWrite(led, HIGH);   // turn the LED on (HIGH is the voltage level)
3. delay(xxxx);                 // wait for a xxxx
4. digitalWrite(led, LOW);    // turn the LED off by making the voltage LOW
5. delay(xxxx);                 // wait for a xxxx
```

TASK 4 Writing, compiling and uploading a Morse code program

Let's now write a simple sketch for flashing a Morse code:

```
int pin = 13;
void setup()
{
  pinMode(pin, OUTPUT);
}

void loop()
{
  dot(); dot(); dot();
  dash(); dash(); dash();
  dot(); dot(); dot();
  delay(3000);
}
void dot()
{
  digitalWrite(pin, HIGH);
  delay(250);
  digitalWrite(pin, LOW);
  delay(250);
}
void dash()
{
  digitalWrite(pin, HIGH);
  delay(1000);
  digitalWrite(pin, LOW);
```

```
  delay(250);
}
```

If you run this sketch, it should flash out the code for **SOS** (a distress call; **Save Our Souls**) on pin 13.

Fritzing Overview

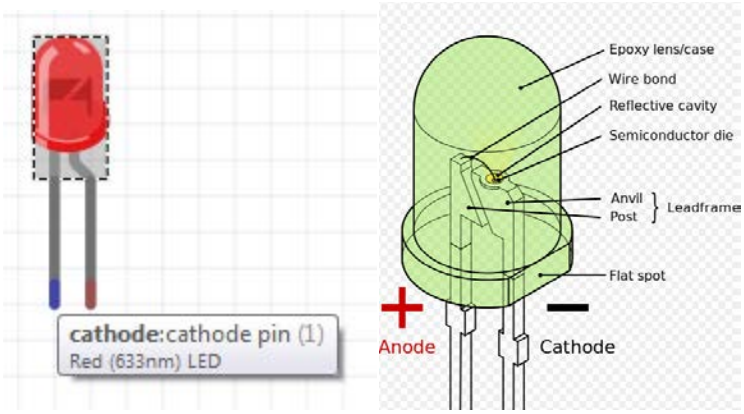
Fritzing is a free open source (they do ask for a contribution) drawing tool with stock libraries of common devices and components used with our Arduino workshop.

- Sign up for an account on the Fritzing site.
- Download the Fritzing software from http://fritzing.org/download/ about 50MB.

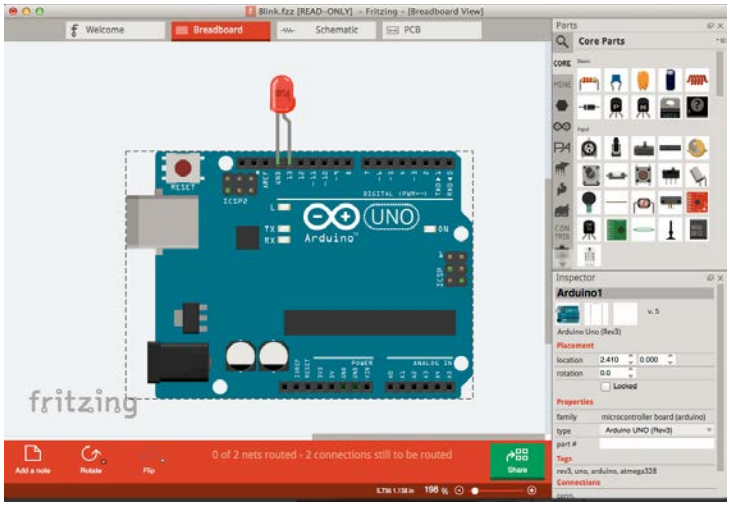
TASK 1. Installing Fritzing

Please make sure your system satisfies one of these requirements:
Windows - XP and up
Mac - OSX 10.7 and up, though 10.6 might work too
Linux - a fairly recent linux distro with libc >= 2.6

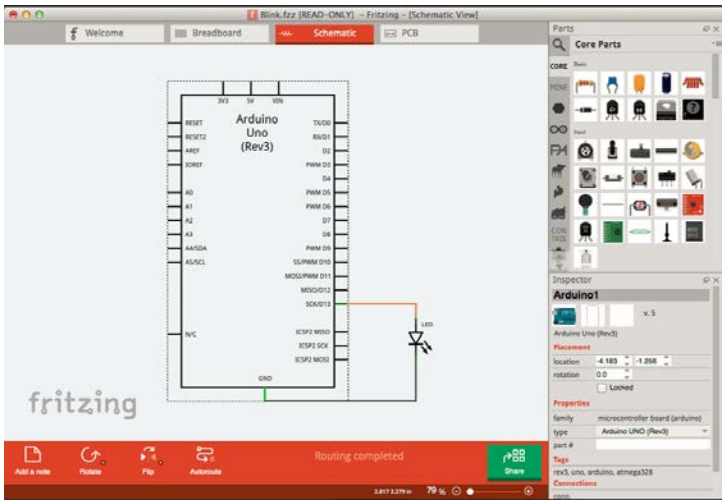
1. Start downloading the Fritzing package that's right for you.
2. Unzip your Fritzing folder somewhere convenient on your hard drive.
    - This may also be a good time for you to create a shortcut to the Fritzing application.
3. To start Fritzing:
    - on *Windows*: double-click fritzing.exe
    - on *Mac*: double-click the Fritzing application
    - on *Linux*: double-click Fritzing, or try ./Fritzing in your shell window
    - **Draw your UNO layout in the Fritzing application.**
    - **Note the LED polarity in Fritzing from the post and anvil supports or mouse over the component to view the lead polarity**



If you have issues, read the Notes or Blogs on the Fritzing site.



Example of a Fritzing Breadboard layout



Example of a Fritzing Schematic layout

We will use Fritzing during this workshop to:

- Create a wiring layout drawing of our Uno and any components we use with the solderless breadboard
- Create a schematic diagram from the wiring layout above
- Create a printed circuit board (PCB) file from the schematic and output a Gerber file to be used to manufacture the PCB.

Weather Shield: Physical and Electrical Overview

https://learn.sparkfun.com/tutorials/weather-shield-hookup-guide

The Weather Shield we will use is an Arduino UNO shield that provides barometric pressure, relative humidity, luminosity, and temperature. There are also connections to optional sensors such as wind speed, wind direction, a rain gauge and GPS for location and timing.

For this workshop, we are interested in the relative humidity, luminosity, temperature, and wind speed and wind direction sensors.

Normally, the shield comes without headers that you will need to solder. A trick is to use the Arduino Uno board headers to hold the shield headers steady while you solder. ***This step has been done for you with this workshop.*** You will also need two 6 pin RJ11 connectors should you wish to use the cables from the Anemometer. If you choose to remote your weather station, you can use a wireless solution such as XBee series 1, which will be shown in the workshop.

Things you should know about this weather shield and can find from the **manufacturers data sheets URL's below**:

The HTU21D humidity sensor, MPL3115A2 barometric pressure sensor, and ALS-PT19 light sensor.

- **Create a weather shield diagram using Fritzing to identify components:**
- **Identify the humidity, barometric pressure and light sensor components.**

The connector for the GP-635T compact GPS module

- **Identify the connector**

The optional connectors for the SparkFun weather meters

- **Identify those connectors**

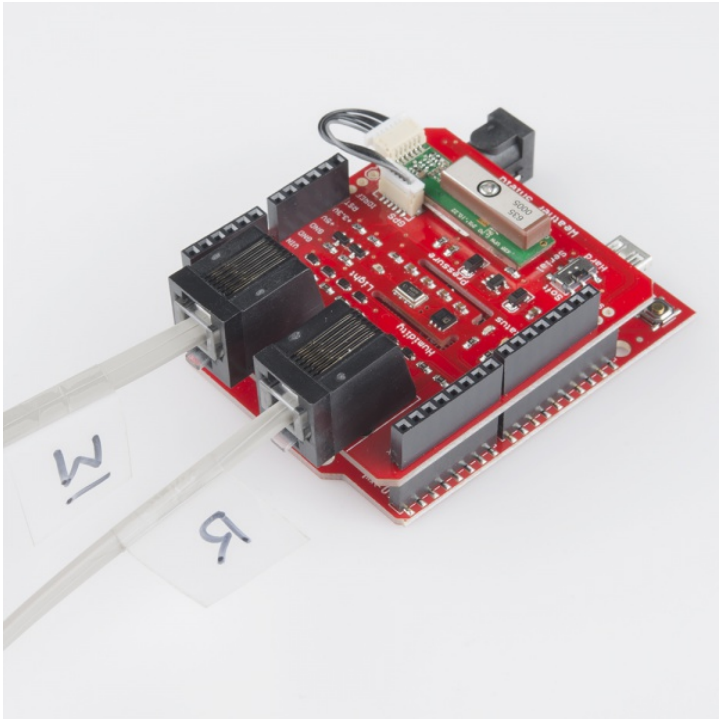
The Weather shield can operate from 3V to 10V and has built in voltage regulators and signal translators

- **Identify the regulator**

Typical humidity accuracy of ±2%

Typical pressure accuracy of ±50Pa

Typical temperature accuracy of ±0.3C



Shield on a RedBoard with optional weather meter ('W'ind and 'R'ain cables) and GPS attached

Datasheets

There's a lot of technology on this shield. Here are the datasheets for reference:

- HTU21D Humidity
- MPL3115A2 Pressure
- ALS-PT19 Light
- GP-635T GPS
- Weather Meters
- HTU21D Humidity Repo and Library
- MPL3115A2 Pressure Repo and Library

- If you're interested in using GPS with Arduino definitely checkout Mikal Hart's TinyGPS++ library
- Consider adding an OpenLog for datalogging the weather readings over time
- Electric Imp is a good way to add WiFi to get a truly wireless weather station

Reading Assignment for next week.

- **I^2C Protocol**
- **Installing an Arduino library**
- **How to install Arduino shield headers**
- **What are pull-up resistors?**
- **HTU21D Humidity Sensor Hookup Guide**

WEEK 2

Objectives for this week:

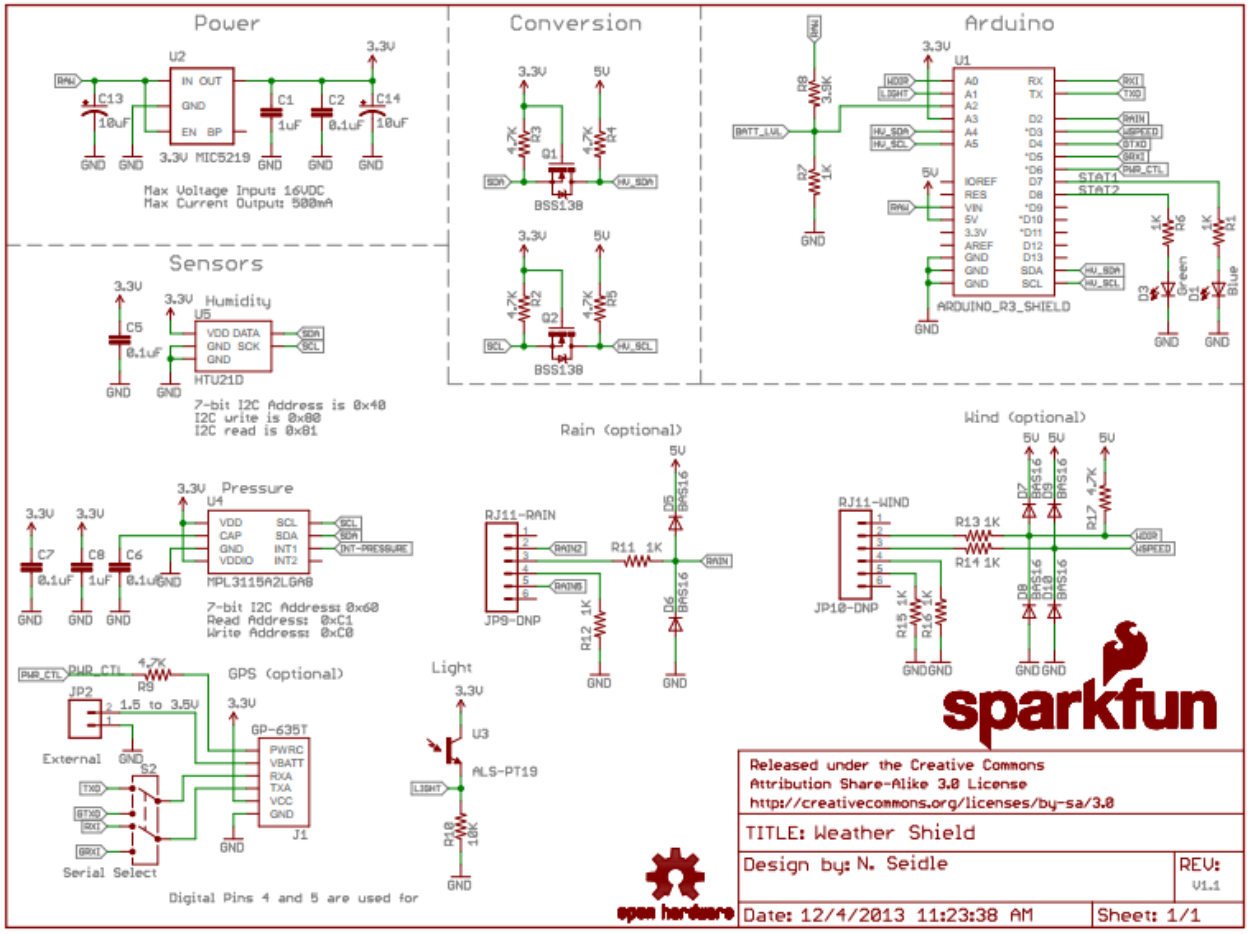
- The Sparkfun Weather shield and how it works.
- Understanding the Sparkfun weather shield electrical schematic
- The purpose of Libraries in software coding.
- Writing Arduino code for the weather shield

Weather Sensors and how they communicate.

Given that not all sensors use the same technology, voltage or communication protocols, understanding how various sensors work is necessary if you wish to build your own apparatus.

The Inter-integrated Circuit (I^2C) Protocol is a protocol intended to allow multiple "slave" digital integrated circuits ("chips") to communicate with one or more "master" chips. Like the Serial Peripheral Interface (SPI), it is only intended for short distance communications within a single device. Like Asynchronous Serial Interfaces (such as RS-232 or UARTs), it only requires two signal wires to exchange information.

Weather Shield schematic



Weather Shield Schematic diagram

The purpose of Libraries in software coding

The value of a library is the reuse of the behavior. When a program invokes a library, it gains the behavior implemented inside that library without having to implement that behavior itself. Libraries encourage the sharing of code in a modular fashion, and ease the distribution of the code. The behavior implemented by a library can be connected to the invoking program at different program lifecycle phases. If the code of the library is accessed during the build of the invoking program, then the library is called a static library. An alternative is to build the executable of the invoking program and distribute that, independently from the library implementation. The library behavior is connected after the executable has been invoked to be executed, either as part of the process of starting the execution, or in the middle of execution. In this case the library is called a

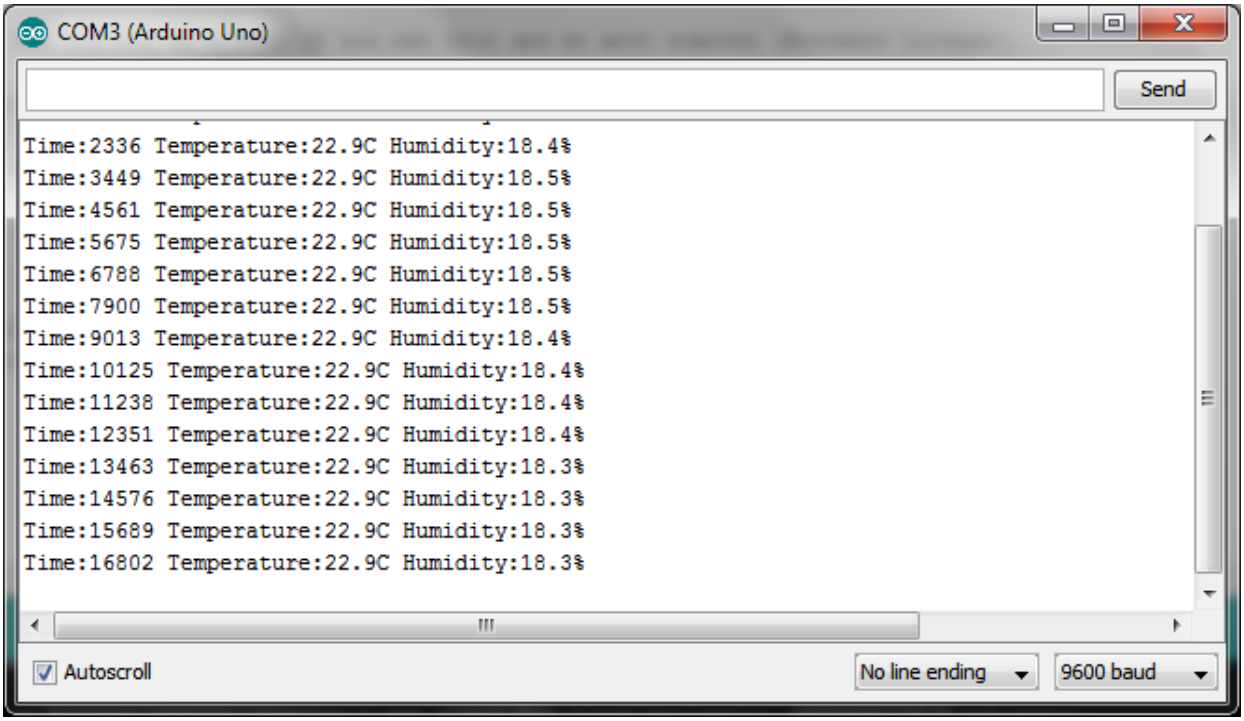
dynamic library. A dynamic library can be loaded and linked as part of preparing a program for execution, by the linker. Alternatively, in the middle of execution, an application may explicitly request that a module be loaded.

Example Firmware

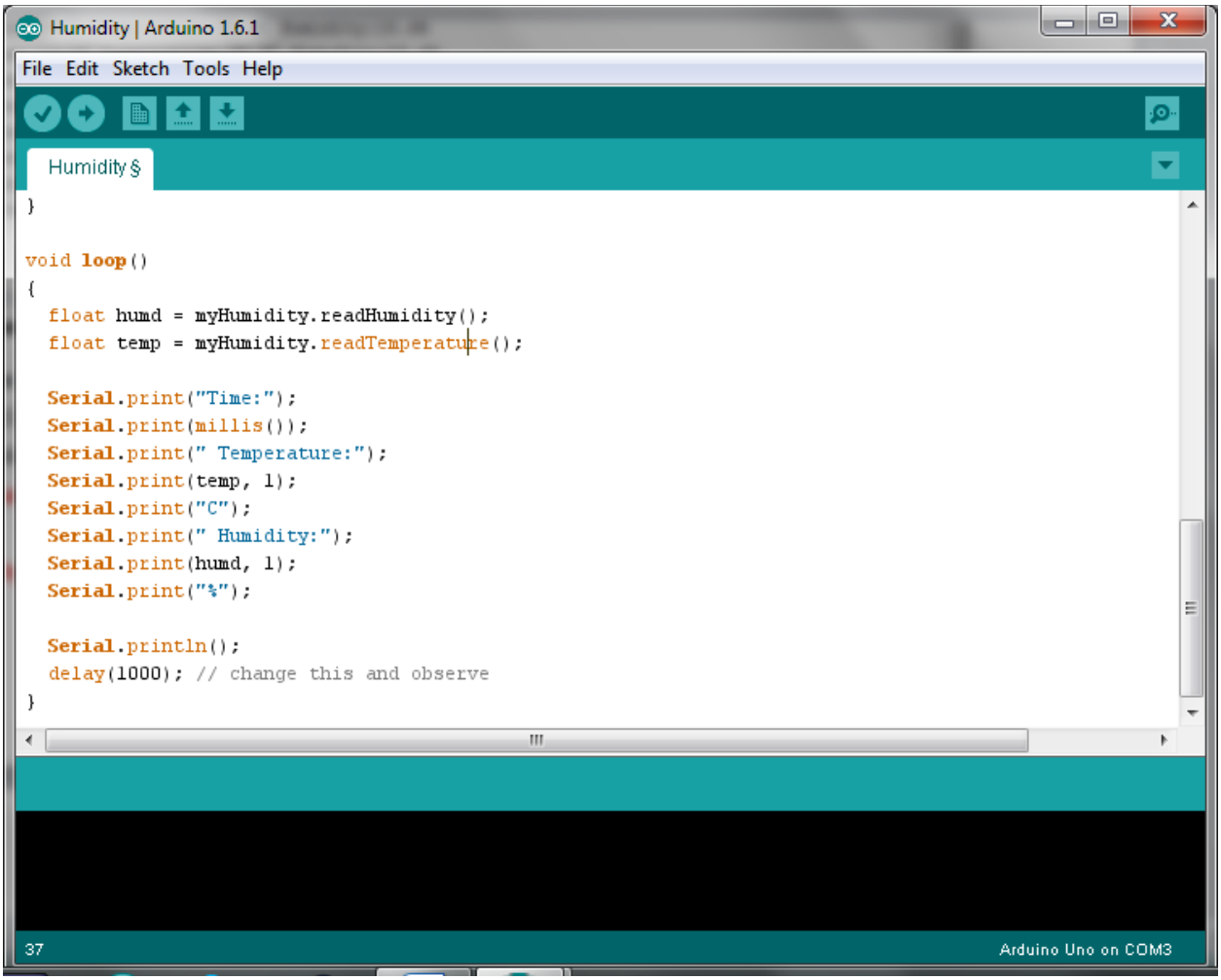
The Weather Shield example relies on the HTU21D and MPL3115A2 libraries.

Download the libraries here then install them into your Documents/Arduino folder.

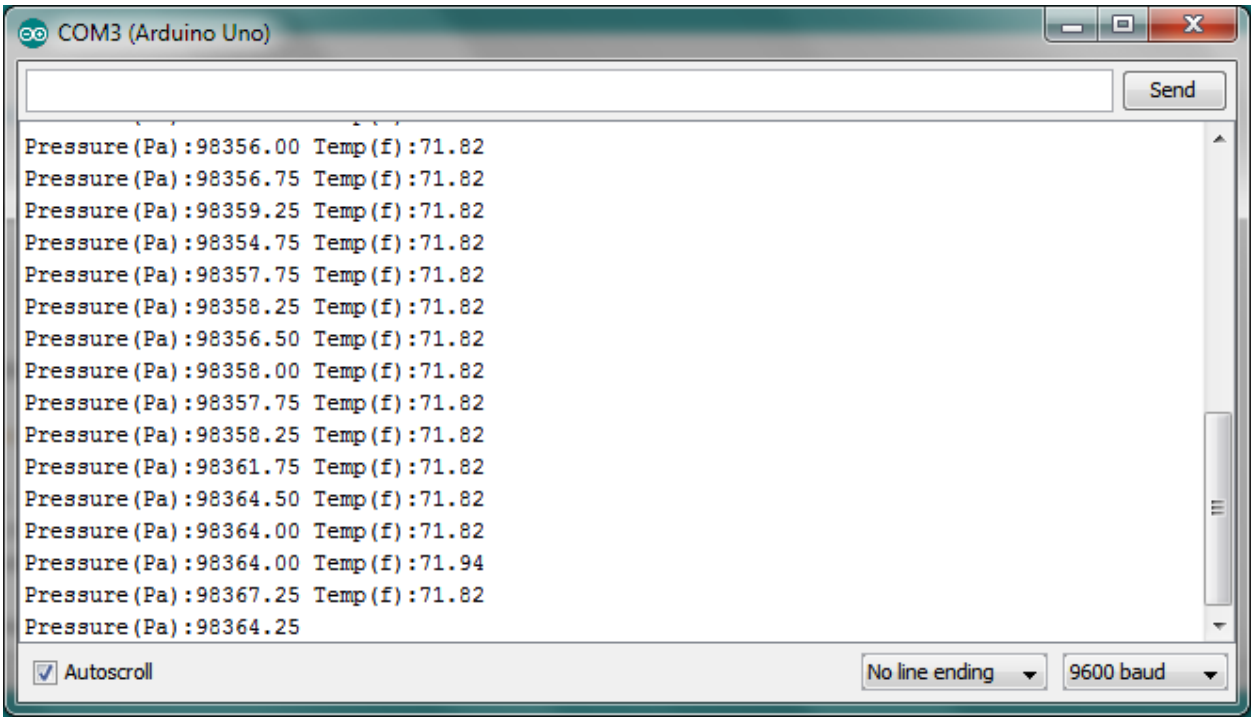
Open the example sketch from the HTU21D folder (there should be a subfolder in each of the library folders you downloaded called examples, open up the example in there), and load it onto your Arduino. Open the serial monitor (button is at the top on the far right that looks like a magnifying glass) at 9600bps. You should see an output string every second containing the humidity and temperature:



Now let's try changing how often data is output to the monitor. Close the serial monitor and look at the example code. Look at the line that says delay(1000) (shown in the image below). Remember this from week 1? This line delays how often the output is printed to the serial monitor. Change this value to any value you want and notice what happens.



Now open the Pressure example from the MPL3115A2 folder. Compile and upload it to the Arduino and open up the serial monitor. You should see the barometric pressure and the temperature in degrees Fahrenheit similar to this output:



Now let's modify the humidity example to include temperature, barometric pressure and humidity. Open the humidity example from the HTU21D folder. Click file and then save as and then name it to whatever you like and save it to wherever you like.

Now let's go in depth in examining the code. Look at the top of the code. You should see this:

```
#include <Wire.h>
#include "HTU21D.h"

//Create an instance of the object
HTU21D myHumidity;
```

This is where we import header files for our code. These header files come from installed libraries whether it's any installed libraries or libraries that come with the Arduino IDE. They contain the functions we will use for our code so we don't have to write them ourselves every time we have to do use that function. Image rewriting a function you will end up using over again each time you write a program that needs it! This is how libraries are useful!

In this example we are importing the wire and the humidity libraries. Notice how in the pressure example it also imports the wire library but instead of humidity it imports the MPL3115A2 library. We want to use both the libraries here. **Type this at the top:**

```
#include "MPL3115A2.h"
```

Now we need to create an instance of this object. See this:

```
//Create an instance of the object
HTU21D myHumidity;
```

Type this below:

```
MPL3115A2 pressure;
```

What we did was create an object of MPL3115A2 called pressure. When we want to use the functions in MPL3115A2 we use pressure followed by a decimal point and then the function we want to use. Refer to the header file or C++ file for the library you are using for the available functions. Note we did not need to call it pressure – we could have called it anything we wanted. For example this is one of the functions we will use for pressure:
pressure.readPressure();

Let's continue modifying the program. Note: You can get rid of the comments at the beginning of this program we are modifying.

Now let's set up the program. We now put code into the void setup() part of the code. It should look like this:

```
void setup()
{
  Serial.begin(9600);
  Serial.println("HTU21D Example!");

  myHumidity.begin();
}
```

Let's get rid of that line that says HTU21D Example! We don't need it for what we are doing.

Now here you see the line Serial.begin(9600); This is the baud rate of the output. Leave this alone. The part that says myHumidity.begin() is the line that gets the humidity sensor working. Now let's add the lines to get the pressure sensor working.

If you look at the pressure sensor example, MPL3115A2 you see this line:

myPressure.begin();

Note that in the example there was two lines, myPressure.setModeBarometer(); and myPressure.enableEventFlags(); We need these too for pressure measuring! The former is to set the sensor to read pressure as it can also read altitude as well and the latter is required in the setup part according to the C++ file in the MPL3115A2.

Type this into your program:

```
pressure.setModeBarometer();
pressure.enableEventFlags();
pressure.begin();
```

Your setup function should now look like this:

```
void setup()
{
  Serial.begin(9600);
  pressure.setModeBarometer();
  pressure.enableEventFlags();
  pressure.begin();
  myHumidity.begin();
}
```

Now let's display the readings. Look at the loop section. The code should look like this:

```
void loop()
{
  float humd = myHumidity.readHumidity();
```

```
  float temp = myHumidity.readTemperature();

  Serial.print("Time:");
  Serial.print(millis());
  Serial.print(" Temperature: ");
  Serial.print(temp, 1);
  Serial.print("C");
  Serial.print(" Humidity: ");
  Serial.print(humd, 1);
  Serial.print("%");

  Serial.println();
  delay(1000); // change this and observe
}
```

If you want to get rid of the lines Serial.print("Time:"); and Serial.print(millis()); as we don't need them at this time. Also add spaces after Temperature and Humidity for readability when we look at the output.

As you can see here the humidity and temperature was already done. Let's add the pressure readings to this but first let's go over some of this code.

```
float humd = myHumidity.readHumidity();
float temp = myHumidity.readTemperature();
```

These lines assign values to the float variable. What we see here is the object myHumidity is calling the function readHumidity() and readTemperature() and storing them into float variables. Looking at the C++ or the header file for the pressure sensor we can see that there is a function that we can use called readPressure().

Note: Due to an error with the readTemperature() function constantly incrementing the temperature in output we are going to change this line. Delete it and replace with this:

```
float temp = pressure.readTemp();
```

Notice how there is longer any highlighting.

Declare and initialize a variable that will store the float value returned from the readPressure() function.

Note we can give variable names anything we want but let's call it pres.
The top part of the code should now look like this:

```
float humd = myHumidity.readHumidity();
float temp = pressure.readTemp();
float pres = pressure.readPressure();
```

Now let's look at the part after that where the data gets printed.
We can already see that temperature and humidity is printed already. This one is straight forward when looking at the code that is already there. You can see in the brackets of the Serial.print() function you put the words you want enclosed in double quotation marks. However you do not need quotation marks when you want to print variables. Instead put the variable name with a comma at the end and after the comma put the number of decimal places we want.

Add in the lines that will display "Pressure: " and then the value with one decimal place after the point and then add another line that says "Pascals" or "Pa".

Answer:

```
Serial.print(" Pressure: ");
Serial.print(pres, 1);
Serial.print("Pa");
```

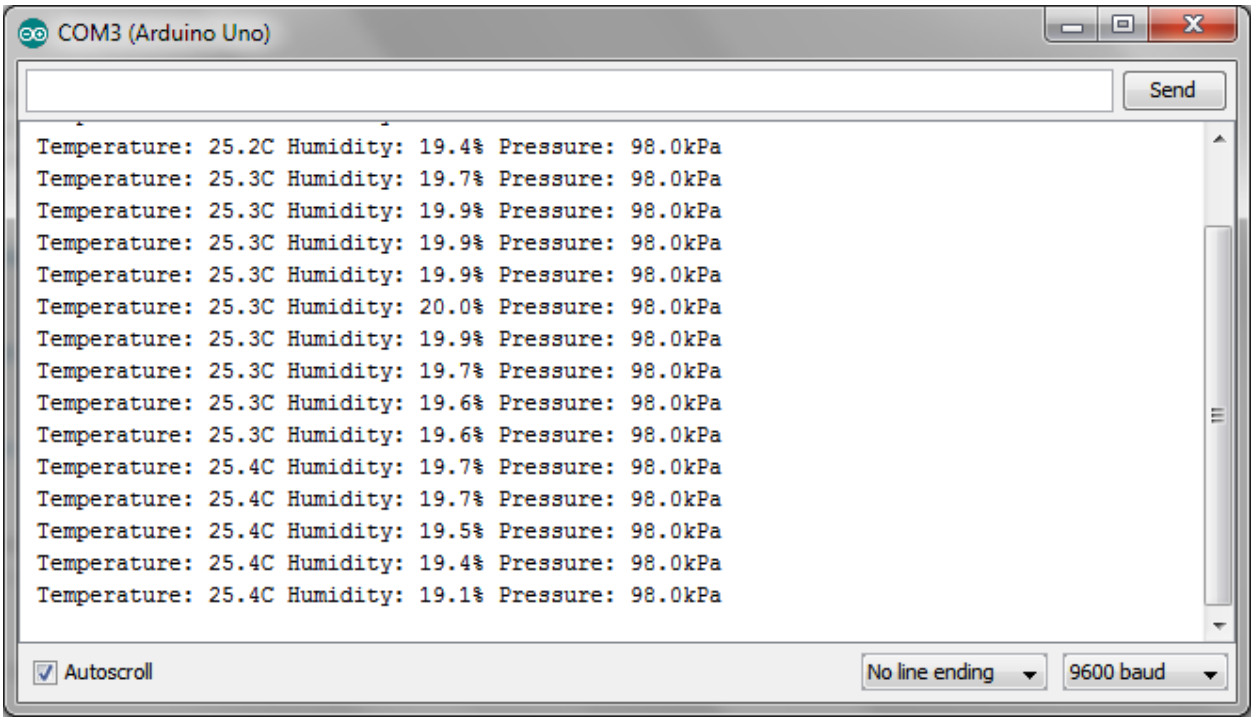
Wasn't hard was it? Now here's a bonus. The pressure sensor will read the pressure in Pascals. The atmospheric pressure is actually so large it is measured in kilopascals. Here's a challenge.

Convert the reading into kilopascals and change the label to kPa or kilopascals.

Answer: We can do this by simply going to where we initialized the variable that stores the pressure and divide it by 1000 or divide it by 1000 when we print it. Example: float pres = pressure.readPressure() / 1000; or Serial.print(pres / 1000, 1); then modify this line **Serial.print(" kPa");**

You can also change the delay line here to whatever you want.

Now compile the program and upload it to the Arduino. The output should look similar to this:



Let's add one more reading to our program, ambient light. This one is tricky as it does not have any libraries that need to be imported. Instead the values get sent to the analog port. We then need to do a calculation on the input to get the calculation.

Let's get started by adding these lines in the void setup() function.

pinMode(A1, INPUT);
pinMode(A3, INPUT);

Before the Serial.begin(9600); line, add this line:

Wire.begin();

The reason why we need this is we need the input for the light sensor and we need to know what the operating voltage is in order to calculate it. Analog pin 1 is for the light sensor and analog pin 3 is for the operating voltage. We need it set these to input so we can read what these values are. Also the light sensor uses I^2C so we need this line. The reason we put it before the Serial.begin(9600) line is to prevent any errors from showing up on the first line. You may have already noticed some when doing the humidity and pressure part.

Now we need to calculate the ambient light value. At the bottom of the void loop() function we will create a function to calculate the ambient light value. **Copy and paste this code:**

```
float readLight()
{
  float operVolt = analogRead(A3); // get the operating voltage of the weather shield
  float lightSen = analogRead(A1); // get input voltage for the light sensor
  operVolt = 3.3 / operVolt;       // divide the reference voltage by the operating voltage
  lightSen *= operVolt;            // multiply this voltage with the light sensor voltage to get
ambient light result
  return(lightSen);
}
```

Here we created a function that will return a float once the ambient light is calculated. What is going on in the function is the value from analog ports 1 and 3 are read and stored into variables, the operating voltage from pin A3 and the light value from pin A1. The value 3.3 (volts) is divided by the operating voltage on pin A3. The resulting value is then multiplied by the light sensor value on A1 to get the final result, which is then returned.

Add a variable and initialize it to read the ambient light in the void loop() function:

Answer:

```
float lght = readLight();
```

So as you may can tell we just added a variable that calls the readLight() function we created. The value that is returned from the function is stored in that variable we created. Remember if the function returns a value (i.e. if the function does not begin with void) it needs to be stored in a variable.

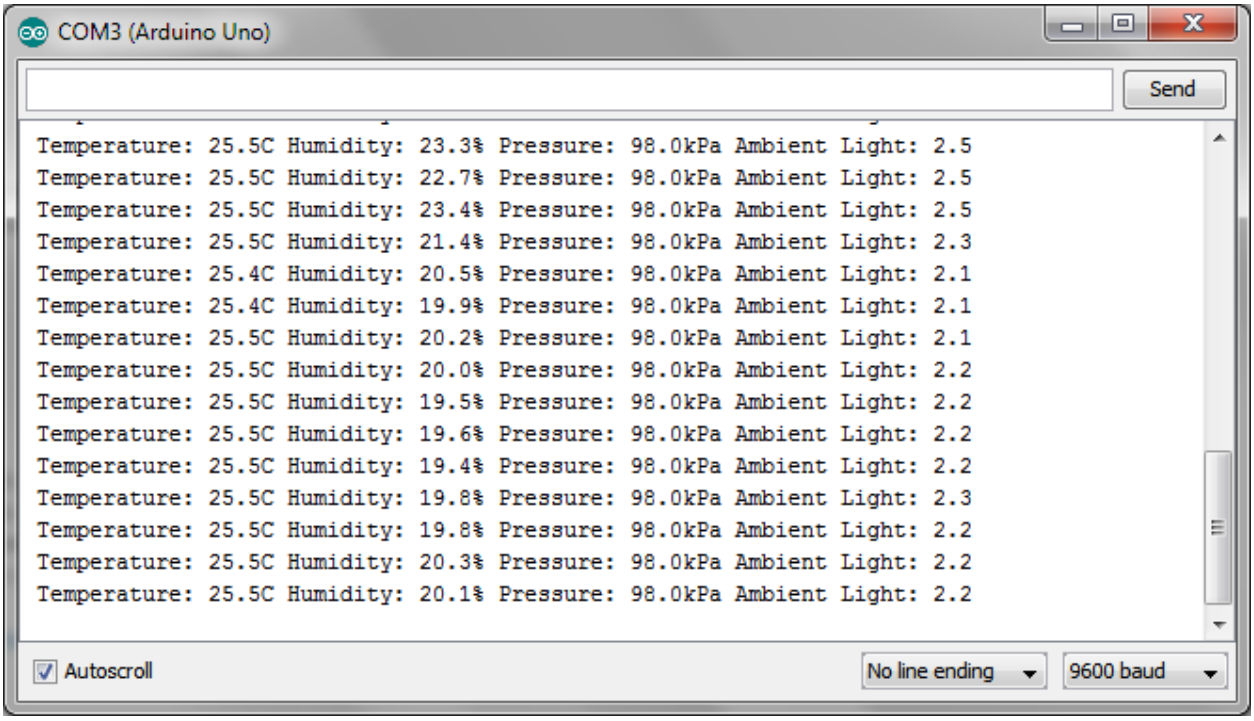
Now we need to add lines that will print out "Ambient Light: " and the ambient light value.

You know how to do this as we did it earlier when we printed out the barometric pressure value.

Answer:

```
Serial.print(" Ambient Light: ");
Serial.print(lght, 1);
```

Now compile and upload the program to the Arduino. The output should look similar to this one: Note that results will vary.

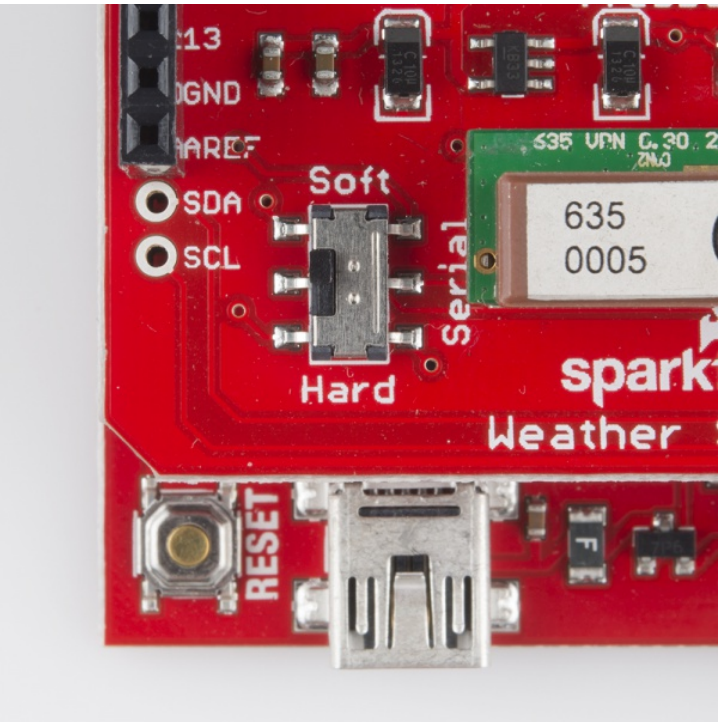


Good job! Later on we will explore how to save this data and display it on a GUI.

If you already have a GPS module, here is an example with GPS installed:
Note: We are going to cover the GPS in depth on Week 7. This is an example of how the GPS works should you have a GPS module at this time.

Shield on a RedBoard with optional weather meter connectors and GPS attached

Attach the GP-635T GPS module using the short cable. To secure the module, there is space on the shield to attach the module using double-stick tape.

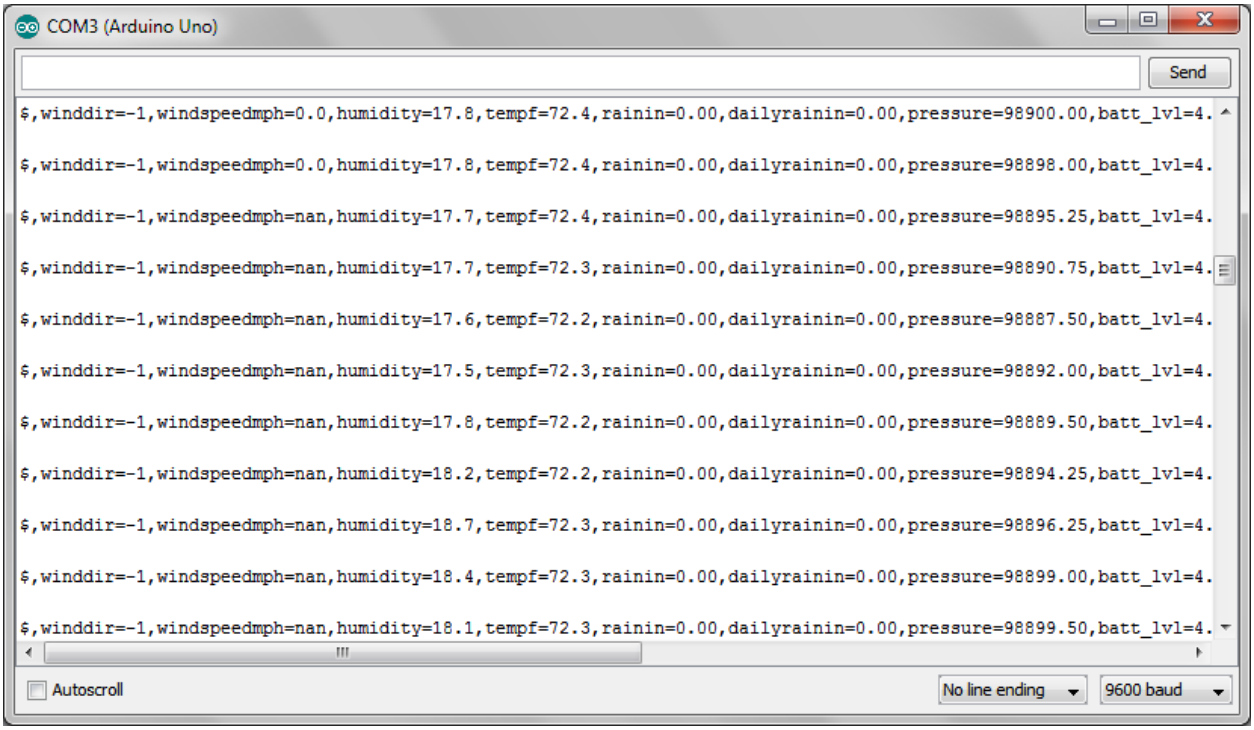


Serial pins are connected to digital pins 4 and 5 when Serial is set to soft and are attached to the internal UART when set to hard.

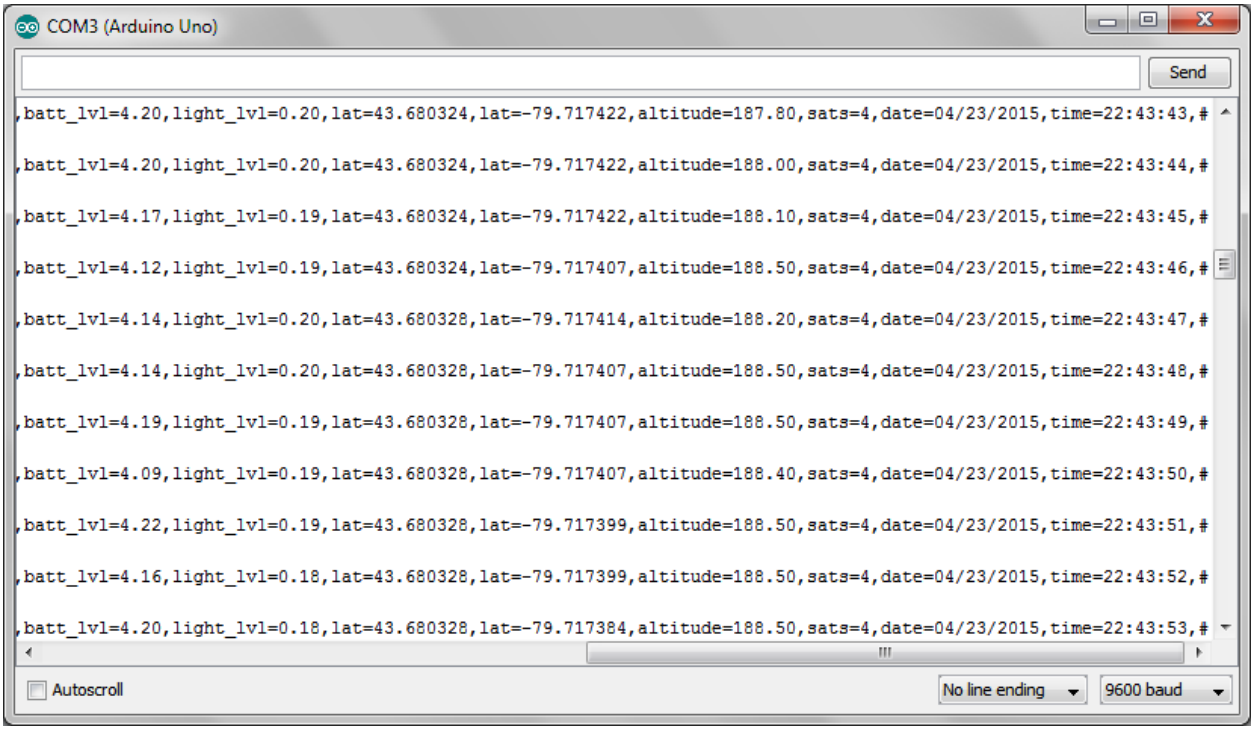
There is a switch labeled **Serial** on the shield. This is to select which pins on the Arduino to connect the GPS to. In almost all cases the switch should be set to 'Soft'. This will attach the GPS serial pins to digital pins 5 (TX from the GPS) and 4 (RX into the GPS).

Grab the GPS example sketch that demonstrates using the GP-635T with all the other sensors. Load it onto your Arduino, and open the serial monitor at 9600. You should see output similar to the following:

Part 1:


COM3 (Arduino Uno)
Send
$,winddir=-1,windspeedmph=0.0,humidity=17.8,tempf=72.4,rainin=0.00,dailyrainin=0.00,pressure=98900.00,batt_lvl=4.
$,winddir=-1,windspeedmph=0.0,humidity=17.8,tempf=72.4,rainin=0.00,dailyrainin=0.00,pressure=98898.00,batt_lvl=4.
$,winddir=-1,windspeedmph=nan,humidity=17.7,tempf=72.4,rainin=0.00,dailyrainin=0.00,pressure=98895.25,batt_lvl=4.
$,winddir=-1,windspeedmph=nan,humidity=17.7,tempf=72.3,rainin=0.00,dailyrainin=0.00,pressure=98890.75,batt_lvl=4.
$,winddir=-1,windspeedmph=nan,humidity=17.6,tempf=72.2,rainin=0.00,dailyrainin=0.00,pressure=98887.50,batt_lvl=4.
$,winddir=-1,windspeedmph=nan,humidity=17.5,tempf=72.3,rainin=0.00,dailyrainin=0.00,pressure=98892.00,batt_lvl=4.
$,winddir=-1,windspeedmph=nan,humidity=17.8,tempf=72.2,rainin=0.00,dailyrainin=0.00,pressure=98889.50,batt_lvl=4.
$,winddir=-1,windspeedmph=nan,humidity=18.2,tempf=72.2,rainin=0.00,dailyrainin=0.00,pressure=98894.25,batt_lvl=4.
$,winddir=-1,windspeedmph=nan,humidity=18.7,tempf=72.3,rainin=0.00,dailyrainin=0.00,pressure=98896.25,batt_lvl=4.
$,winddir=-1,windspeedmph=nan,humidity=18.4,tempf=72.3,rainin=0.00,dailyrainin=0.00,pressure=98899.00,batt_lvl=4.
$,winddir=-1,windspeedmph=nan,humidity=18.1,tempf=72.3,rainin=0.00,dailyrainin=0.00,pressure=98899.50,batt_lvl=4.
Autoscroll
No line ending
9600 baud

Part 2:


COM3 (Arduino Uno)
Send
,batt_lvl=4.20,light_lvl=0.20,lat=43.680324,lat=-79.717422,altitude=187.80,sats=4,date=04/23/2015,time=22:43:43,#
,batt_lvl=4.20,light_lvl=0.20,lat=43.680324,lat=-79.717422,altitude=188.00,sats=4,date=04/23/2015,time=22:43:44,#
,batt_lvl=4.17,light_lvl=0.19,lat=43.680324,lat=-79.717422,altitude=188.10,sats=4,date=04/23/2015,time=22:43:45,#
,batt_lvl=4.12,light_lvl=0.19,lat=43.680324,lat=-79.717407,altitude=188.50,sats=4,date=04/23/2015,time=22:43:46,#
,batt_lvl=4.14,light_lvl=0.20,lat=43.680328,lat=-79.717414,altitude=188.20,sats=4,date=04/23/2015,time=22:43:47,#
,batt_lvl=4.14,light_lvl=0.20,lat=43.680328,lat=-79.717407,altitude=188.50,sats=4,date=04/23/2015,time=22:43:48,#
,batt_lvl=4.19,light_lvl=0.19,lat=43.680328,lat=-79.717407,altitude=188.50,sats=4,date=04/23/2015,time=22:43:49,#
,batt_lvl=4.09,light_lvl=0.19,lat=43.680328,lat=-79.717407,altitude=188.40,sats=4,date=04/23/2015,time=22:43:50,#
,batt_lvl=4.22,light_lvl=0.19,lat=43.680328,lat=-79.717399,altitude=188.50,sats=4,date=04/23/2015,time=22:43:51,#
,batt_lvl=4.16,light_lvl=0.18,lat=43.680328,lat=-79.717399,altitude=188.50,sats=4,date=04/23/2015,time=22:43:52,#
,batt_lvl=4.20,light_lvl=0.18,lat=43.680328,lat=-79.717384,altitude=188.50,sats=4,date=04/23/2015,time=22:43:53,#
Autoscroll
No line ending
9600 baud

Note: The batt_lvl is indicating 4.08V. This is correct and is the actual voltage read from the Arduino powered over USB. The GPS module will add **50-80mA** to the overall power consumption. If you are using a long or thin USB cable you may see significant voltage drop similar to this example. There is absolutely no harm in this. The Weather Shield runs at 3.3V and the Arduino will continue to run just fine down to about 3V. The reading is very helpful for monitoring your power source (USB, battery, solar, etc.).

This example demonstrates how you can get location, altitude, and time from the GPS module. This would be helpful with weather stations that are moving such as balloon satellites, AVL, package tracking, and even static stations where you need to know precise altitude or timestamps.

Resources and Going Further

The Weather Shield example firmware outputs regular barometric pressure. This is very different from the pressure that weather stations report. For more information, see the definition of "altimeter setting pressure". For an example of how to calculate altimeter setting type barometric pressure see the MPL3115A2 hook-up guide. Also checkout the MPL3115A2 library, specifically the `BarometricHgInch` example.

WEEK 3

Objectives for this week:

- Assembling the Anemometer.
- Connecting the Anemometer to the Sparkfun weather shield.

Weather Sensor Assembly p/n 80422 Imported by Argent Data Systems

Usage Notes

This kit includes a wind vane, cup anemometer, and tipping bucket rain gauge, with associated mounting hardware. These sensors contain no active electronics, instead using sealed magnetic reed switches and magnets to take measurements. A voltage must be supplied to each instrument to produce an output.

Assembly

The wind sensor arm mounts on top of the two-piece metal mast and supports the wind vane and anemometer. A short cable connects the two wind sensors. Plastic clips on the underside of the arm hold this cable in place. Screws are provided to secure the sensors to the arm.

Rain gauge

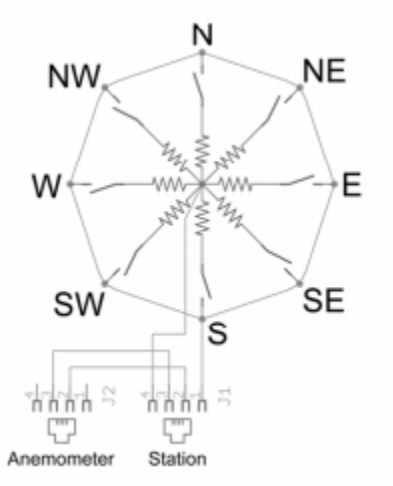
The rain gauge may be mounted lower on the mast using its own mounting arm and screw, or it may be mounted independently. Rain Gauge The rain gauge is a self-emptying tipping bucket type. Each 0.011" (0.2794 mm) of rain causes one momentary contact closure that can be recorded with a digital counter or microcontroller interrupt input. The gauge's switch is connected to the two center conductors of the attached RJ11-terminated cable.

Anemometer

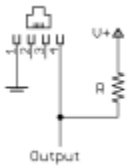
The cup-type anemometer measures wind speed by closing a contact as a magnet moves past a switch. A wind speed of 1.492 MPH (2.4 km/h) causes the switch to close once per second. The anemometer switch is connected to the inner two conductors of the RJ11 cable shared by the anemometer and wind vane (pins 2 and 3.)

Wind Vane

The wind vane is the most complicated of the three sensors. It has eight switches, each connected to a different resistor. The vane's magnet may close two switches at once, allowing up to 16 different positions to be indicated. The switch and resistor arrangement is shown in the diagram below.



An external resistor can be used to form a voltage divider, producing a voltage output that can be measured with an analog to digital converter, as shown below.



Resistance values for all 16 possible positions are given in the table. Resistance values for positions between those shown in the diagram are the result of two adjacent resistors connected in parallel when the vane's magnet activates two switches simultaneously.

Direction (Degrees)	Resistance (Ohms)	Voltage (V=5v, R=10k)
0	33k	3.84v
22.5	6.57k	1.98v
45	8.2k	2.25v
67.5	891	0.41v
90	1k	0.45v
112.5	688	0.32v
135	2.2k	0.90v

157.5	1.41k	0.62v
180	3.9k	1.40v
202.5	3.14k	1.19v
225	16k	3.08v
247.5	14.12k	2.93v
270	120k	4.62v
292.5	42.12k	4.04v
315	64.9k	4.78v
337.5	21.88k	3.43v

The coding part

First let's cover how the wind speed is measured as we will take the programming with the Arduino board further this week.

Let's begin by talking about **interrupts**. An interrupt is a special event that occurs when an event comes along that needs the attention of the CPU right away. The CPU stops what it's doing and then executes certain code to handle the interrupt then returns to doing whatever it was doing before the interrupt occurred. Interrupts can get complicated sometimes, especially in assembly language programming. Fortunately what we need to do for the wind speed sensor is not that hard and actually easy to understand for beginners. And don't worry about assembly language programming – we're not even using it!

Let's start by explaining how to do interrupts with the Arduino boards we are using.

First the variables: Any variable that is going to be modified during an interrupt should be declared by putting volatile followed by the variable type, name, and value. It may not be necessary every time you have an interrupt to have volatile variables in your program, however if your interrupt handling function modifies any variables, then that variable must have the volatile keyword at the beginning of the line where you declare it.

Second the interrupt handling function: The interrupt function is the function that will handle the interrupt and once finished will go back to the part of code it was executing before the interrupt happened. These functions can do anything the programmer likes but good practice is that it must be able to do what the programmer wants and then exit the interrupt function as soon as possible. Also these functions must have a return type of void.

Third and last: Specifying what should happen to get the interrupt to occur and turning on interrupts. When you get your interrupt handler working you need to tell the program that this is the function that will be called when the interrupt happens, as well as what priority the interrupt has and what will happen to trigger it. Then in another line you specify that you want to enable interrupts. This must be done in the void setup() function.

Now let's review on what is happening when the program is calculating wind speed.

Every time the anemometer spins, a contact closes on it which will cause an interrupt to happen. During the interrupt the program checks the time between the last time the speed was calculated and the current time. If it was less then, let's say 15 ms then the program exits the interrupt function. This is to eliminate the issue of contact bounce affecting the speed reading as contact bounce will make the anemometer look like it is spinning faster than it should. Otherwise the program will increment the counter that counts how many times the contacts closed. When it calculates the speed it will find out how much time in seconds passed since the last calculation. Then it will divide the number of times the contacts closed since the last calculation divided by the seconds passed since the last calculation times 1.492 miles per contact close per second. It will then reset the contact count and store the current time in ms into a variable for the next time the wind speed is calculated.

Now let's get programming.

We know that during the interrupt there is a counter that counts the number of times the contacts on the anemometer closes. Each time it closes an interrupt occurs that increments the counter and exits. So we need to declare a volatile variable near the top of the program. We should declare the variable type as an integer. Copy:

```
volatile int contacts = 0;
```

Note that you should always initialize variables you declare. Example here shows the variable was set to 0. Another note that you should declare variables near the top of the program outside of any functions that you are going to modify/use in multiple functions. These are called global variables. Here's another one you should copy:

```
long lastCalcTime = 0;
```

This variable will be used to check when the last time the wind speed was calculated.

Next is the interrupt function. This one is simple. We are going to check if the last time the interrupt occurred was greater than 15 ms ago then increment the contacts count. You can declare functions anywhere in the program except for inside them.

```
void isrSpeed()
{
  // checking if time passed between now and the last calculation is greater than 15 ms
  // to prevent the program from reading the bouncing on the switches
  if (millis() - lastCalcTime > 15)
  contacts++; // increment counter
}
```

Now it's on to the placing of the interrupt. In the void setup() function you place a line calling the function attachInterrupt(). Inside the brackets you put any number you like as the priority number, then the function to execute when the interrupt occurs, and then HIGH, for if the signal is 1, LOW for if the signal is 0, RISING for if the signal goes from 0 to 1, or FALLING for if the signal is going from 1 to 0. We want it to occur after the switch leaves the contact so we put FALLING. After that we just say interrupts() to turn on interrupts for our program. There is also a function that turns off interrupts called nointerrputs(). We won't be using it. Paste this into the void setup() function:

```
attachInterrupt(1, isrSpeed, FALLING);
interrupts();
```

Before we do anything else we need to tell the Arduino that we want to use the pin the speed is coming from for input. We also need to specify that this pin also needs to be connected to a pull-up resistor to prevent damage to the pin. Put this code in the void setup() function:

```
pinMode(3, INPUT_PULLUP);
```

Now let's do the calculations. We already know what the program will do so we write a function that will calculate the wind speed.

```
float calculateWindSpeed()
{
   // find how much time has passed since last calculation
   float timeChange = (millis() - lastCalcTime) / 1000
   // calculate wind speed in km/h
   float windSpeed = (((float)contacts / timeChange) * 1.492) * 1.60934;
   contacts = 0; // reset contacts count
   lastCalcTime = millis();  // store current time into last cacl time variable
   return windSpeed;
}
```

Note: Once we do the calculation of the wind speed we then convert the units into km/h by multiplying by 1.60934. This is of course optional but we will use km/h from hereon.

Finally we then print the variable. Go to the void loop() section and declare a float variable and name it what you like. Set it to call the function we created. Example/copy:

```
float wspd = calculateWindSpeed();
```

Then write a couple of lines that print the wind speed.

```
Serial.print(" Wind Speed: ");
Serial.print(wspd);
Serial.print(" km/h");
```

That's it for the wind speed part. The wind direction part is very short and simple. A brief explanation of the wind vane was given in the beginning of this week. What we do now is in the void loop() where we declared the variable for wind speed declare an int variable for calculating direction like this:

```
int wdir = calculateWindDirection();
```

Then a line that prints the value:

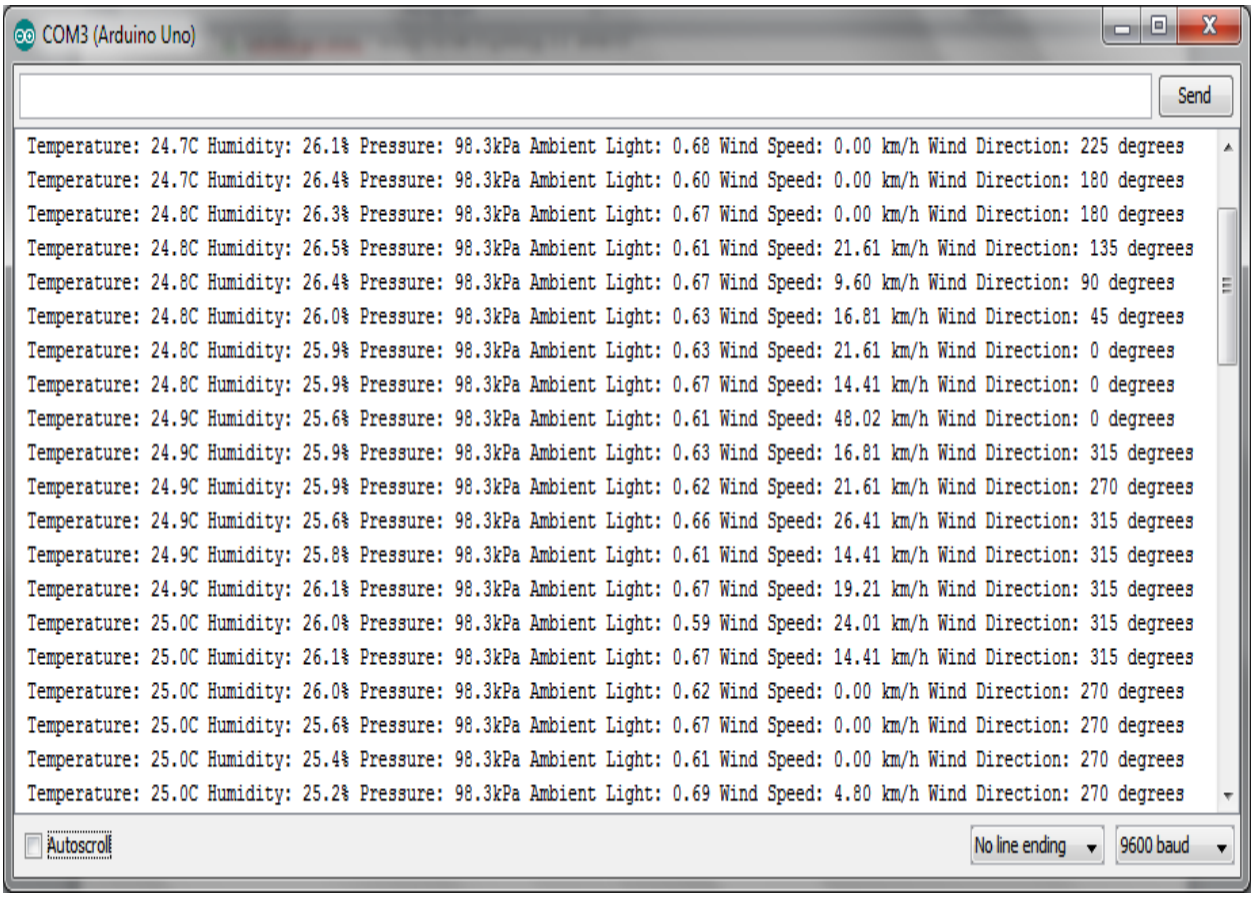
```
Serial.print(" Wind Direction: ");
Serial.print(wdir);
Serial.print(" degrees");
```

Now we create a function that calculates the wind direction Note that this function will return the direction in degrees. Copy:

```
int calculateWindDirection()
{
  unsigned int analogValue;
  analogValue = analogRead(A0);

  if (analogValue < 414) return (90);  // east
  if (analogValue < 508) return (135); // south-east
  if (analogValue < 615) return (180); // south
  if (analogValue < 746) return (45);  // north-east
  if (analogValue < 833) return (225); // south-west
  if (analogValue < 913) return (0);   // north
  if (analogValue < 967) return (315); // north-west
  if (analogValue < 990) return (270); // west

}
```

Let's explain what is going on here. The wind direction value comes into the analog port A0. The program then checks what this value is and then returns the value in degrees. The above code came from the sample code you downloaded in week 2 when we were looking at the GPS. It was modified so that we get only 8 directions, as 16 directions seemed not possible.

Now that you got the code in, your output should look like this:

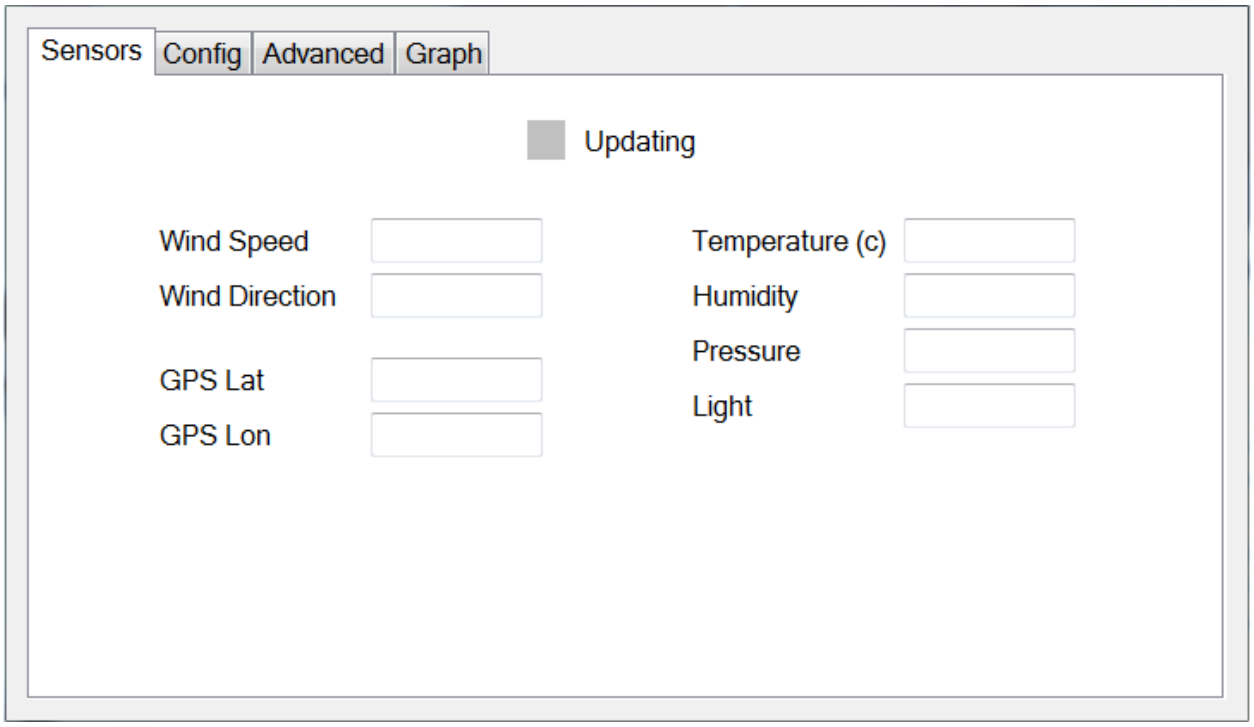


In the next week we will be putting the outputs into a GUI for display.

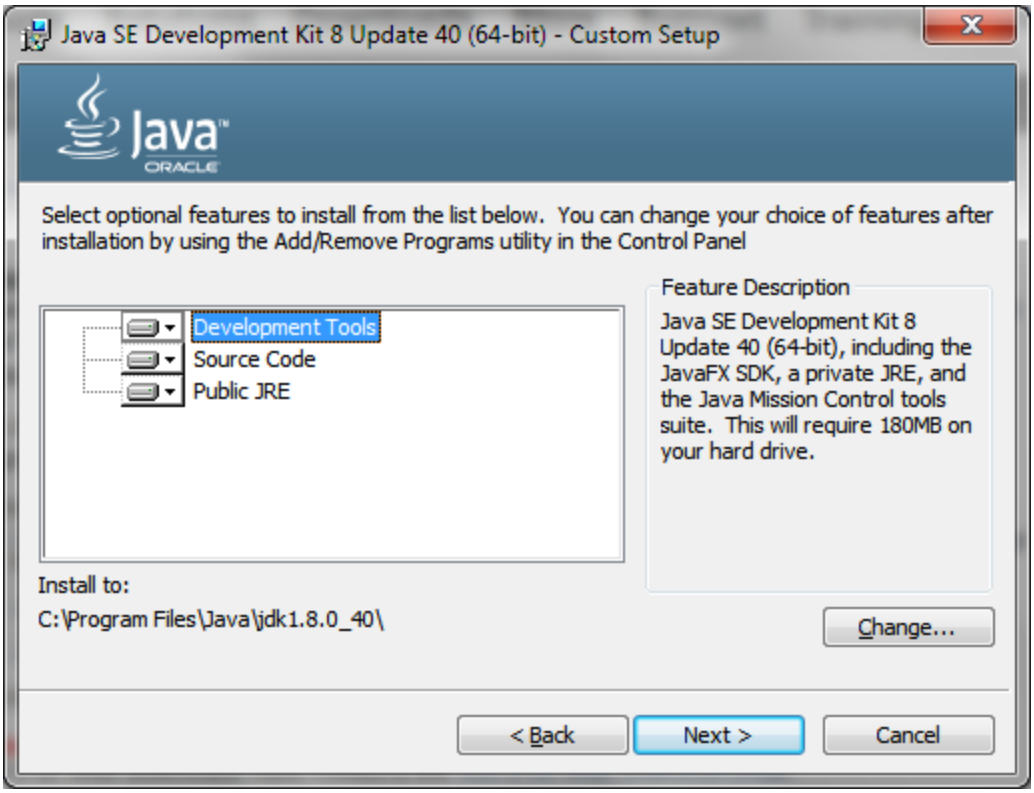
WEEK 4

Objectives for this week:

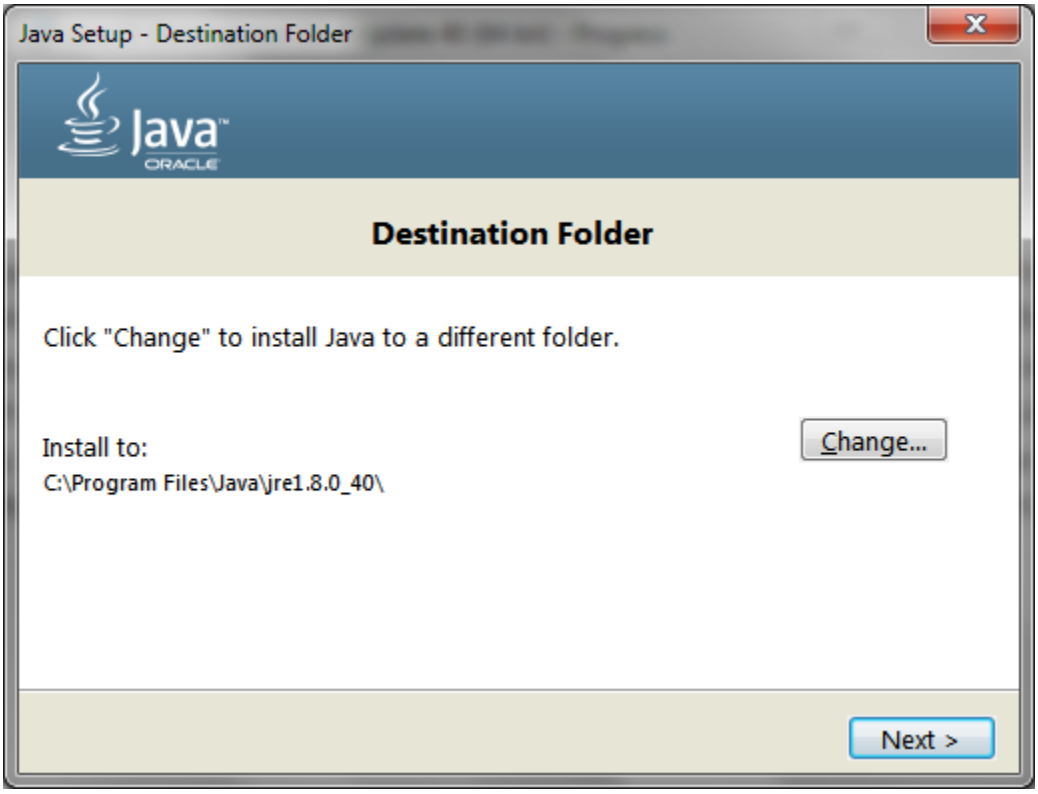
- Displaying weather data in a simple Windows application.



For this week we will need to install several programs. One, we will need to install the Java Development Kit and two; we will need to install Netbeans. Click here to download Java for your computer. Install it and then click this link to go to Oracle's website to download both Java Platform (JDK) and Netbeans. First download and install the Java Platform. Under the Java SE Development Kit, click I agree to the terms and choose the downloader for your system. Open the downloaded file and click next on this window. Leave the settings alone during the installation.



Then click next on this window



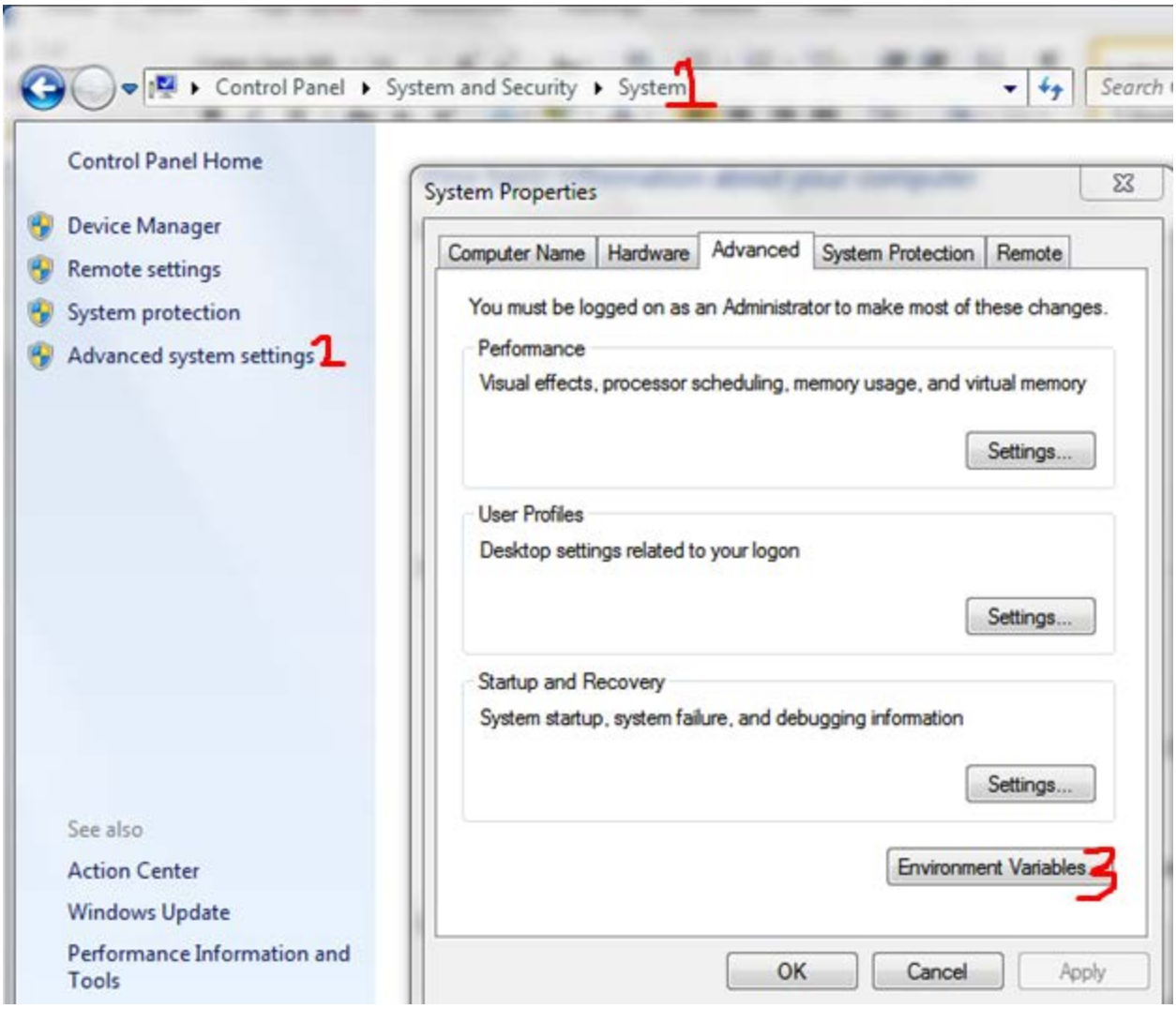
Once the installation is done we need to add the environment variable for your system. If you are using Windows, follow these steps.

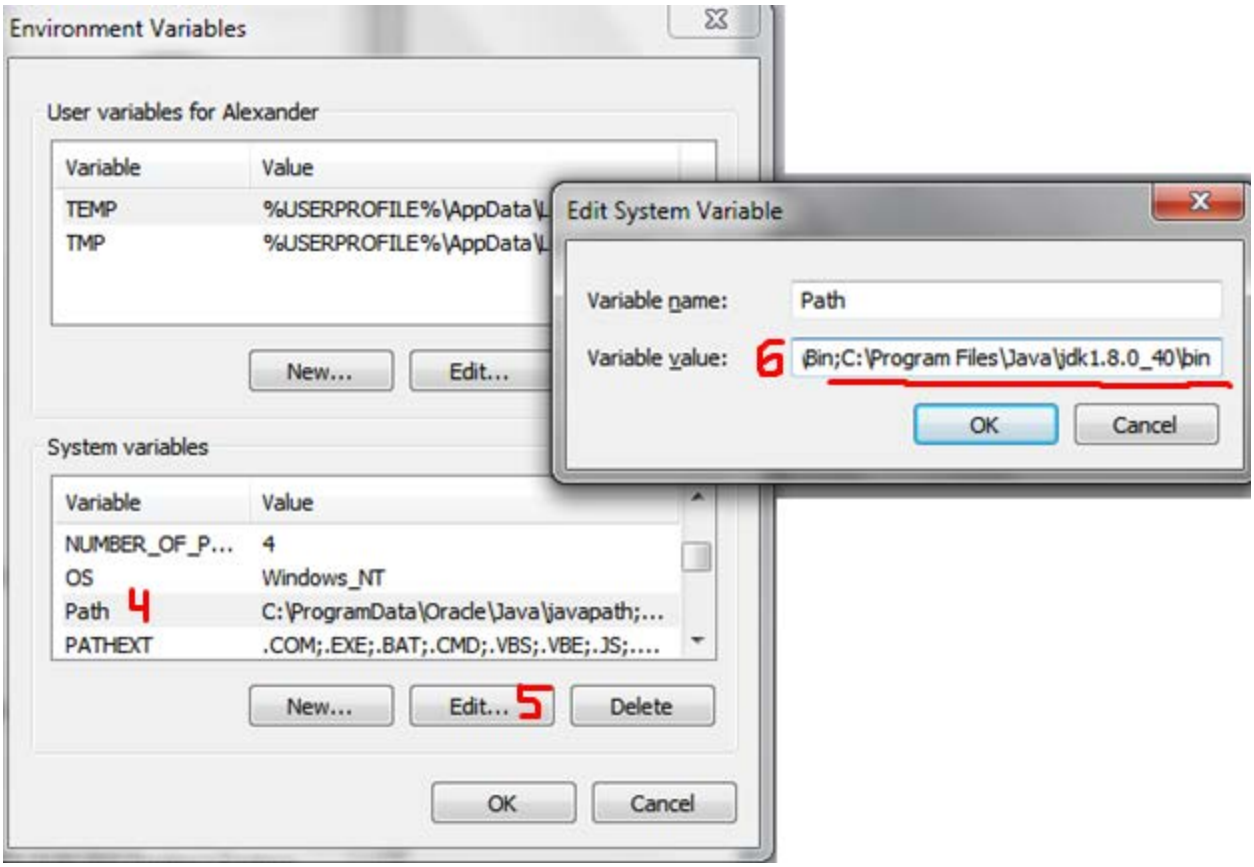
1. Press start, open the control panel and then click System and Security.
2. Click on System, and then look on the left. Click on advanced system settings.
3. On the window that opens click the button that says environment variables.
4. Find the variable that says path and click edit.
5. Scroll to the end of the line that says Variable Value and put a semicolon at the end here.
6. Now you need to find where the Java IDE bin folder is. If you were following the instructions, it should be here:

C:\Program Files\Java\jdk1.8.0_40\bin

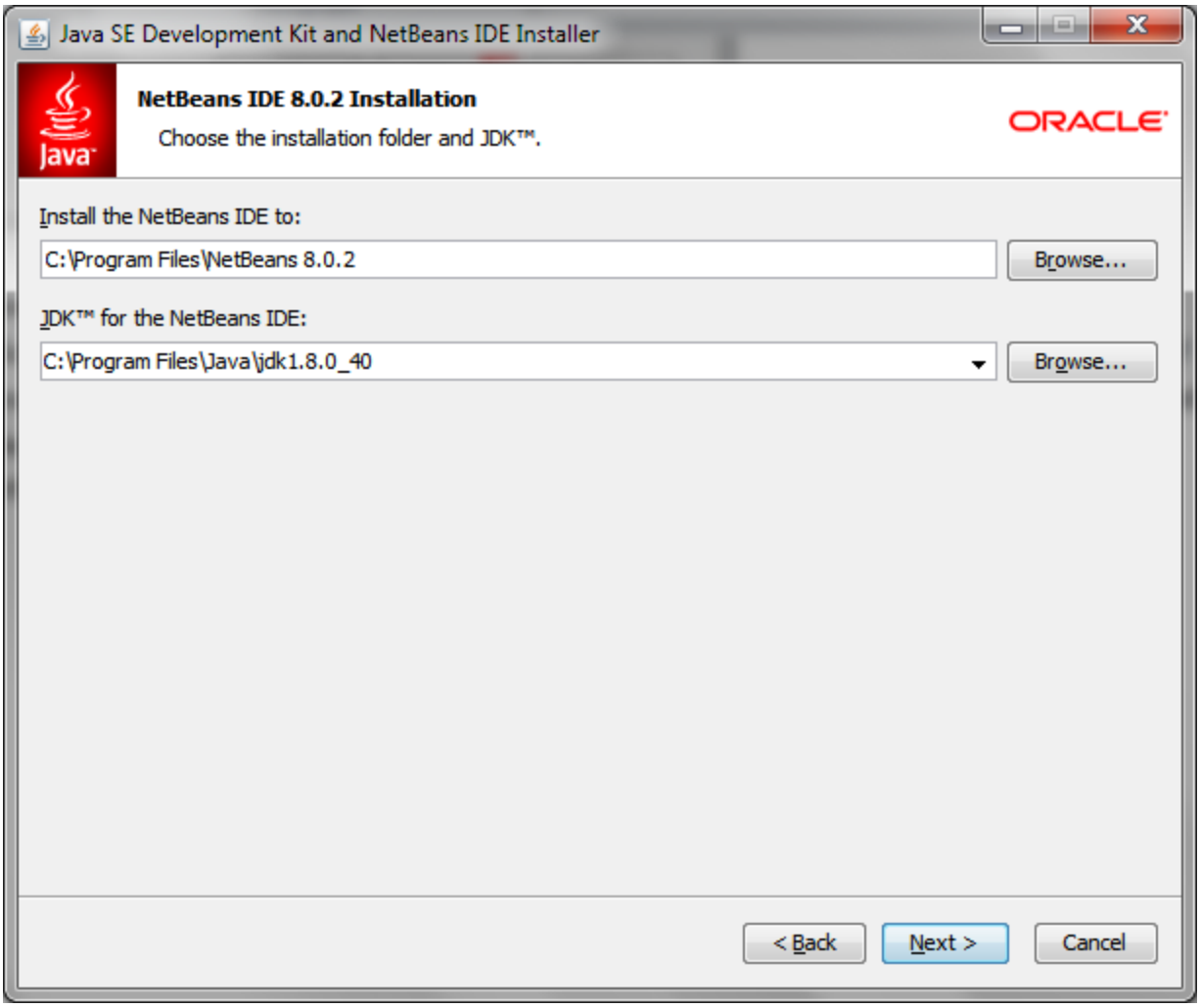
7. Copy and paste the line above after the semicolon you put at the end of the Variable Value line. You do not need a space after the semicolon.
8. Click okay on all of the open windows to confirm.

For reference here are pictures with the path you need to enter entered in the required field.

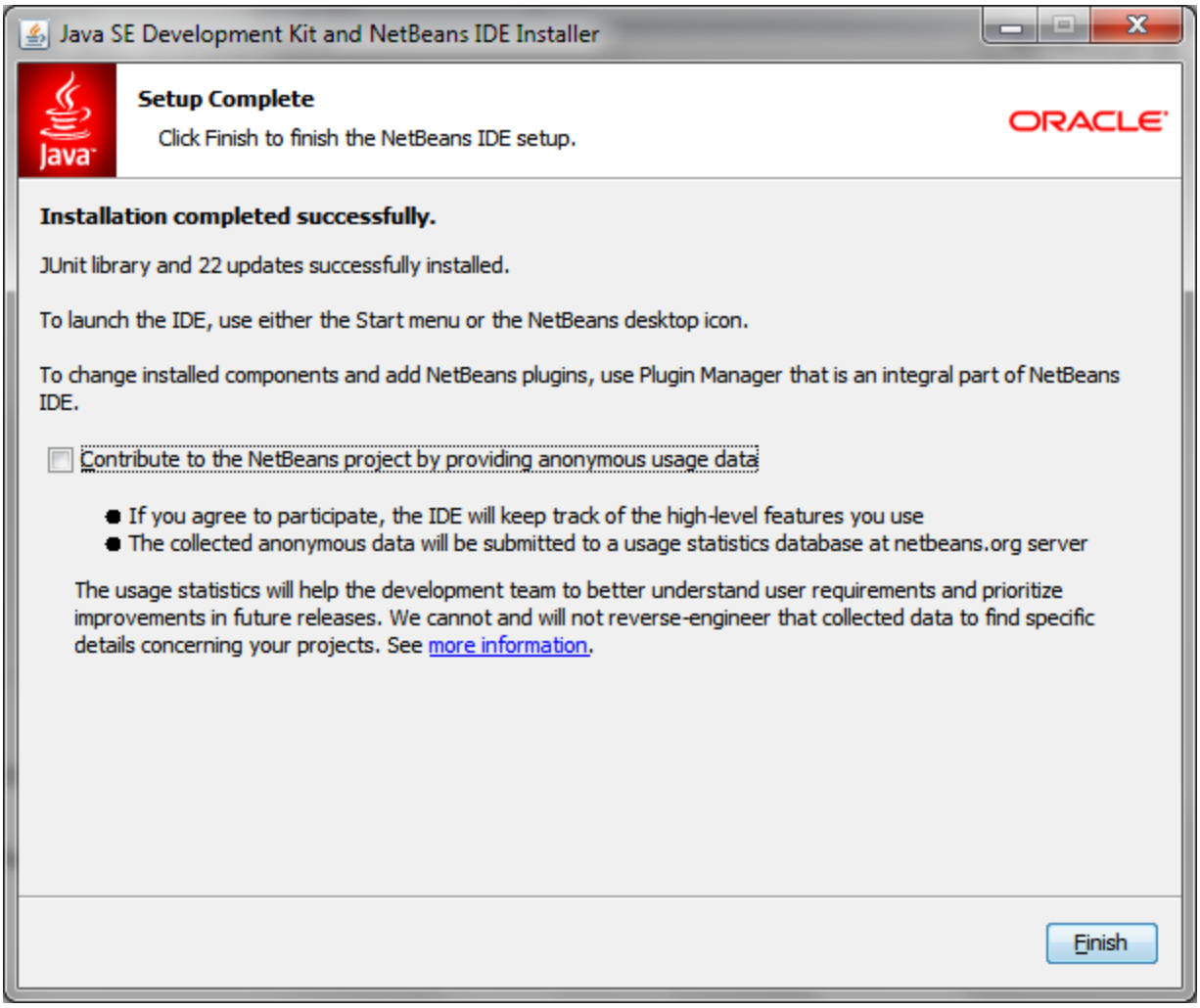

Control Panel ▸ System and Security ▸ System
1
Control Panel Home
Device Manager
Remote settings
System protection
Advanced system settings
1
See also
Action Center
Windows Update
Performance Information and Tools
System Properties
Computer Name
Hardware
Advanced
System Protection
Remote
You must be logged on as an Administrator to make most of these changes.
Performance
Visual effects, processor scheduling, memory usage, and virtual memory
Settings...
User Profiles
Desktop settings related to your logon
Settings...
Startup and Recovery
System startup, system failure, and debugging information
Settings...
Environment Variables
3
OK
Cancel
Apply



Now let's install Netbeans. In case you need to find the page again, here is the link.
Click on Netbeans IDE, then on the next page click I agree, and then select the version of your computer you are downloading for. Once downloaded, run the installer. If you installed the JDK first, which we just did, it will say it is already installed. Click next to continue with installing Netbeans. You don't need to click agree on the next page unless you want the bonus software that we won't need. Then click next on this window.



Then click install on the window after this. Don't change anything during installation. On the next window uncheck the box "Contribute to the Netbeans project […]" and then click finish.



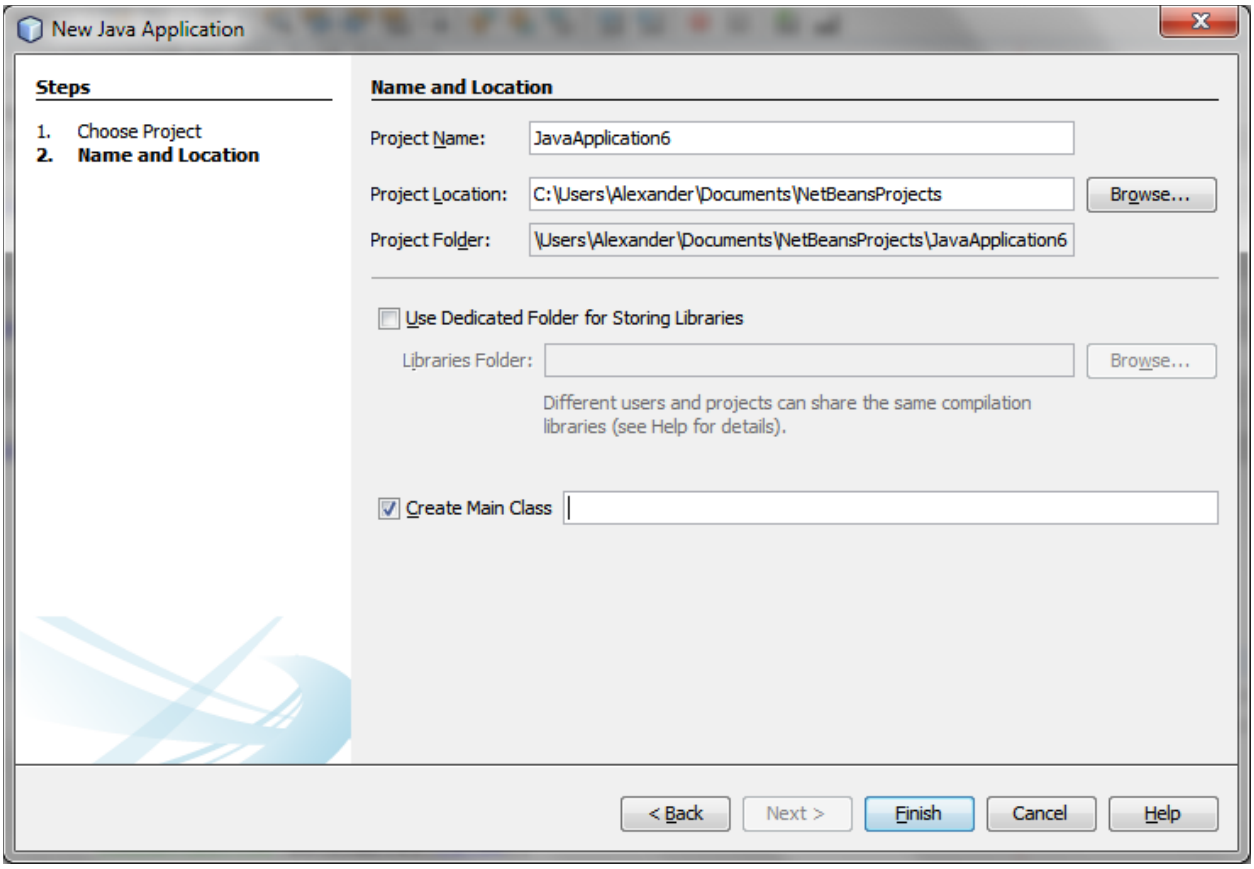
You are now ready to start creating the GUI above!

Let create a new project.

1. Launch the Netbeans IDE and when it open click File and then New Project.

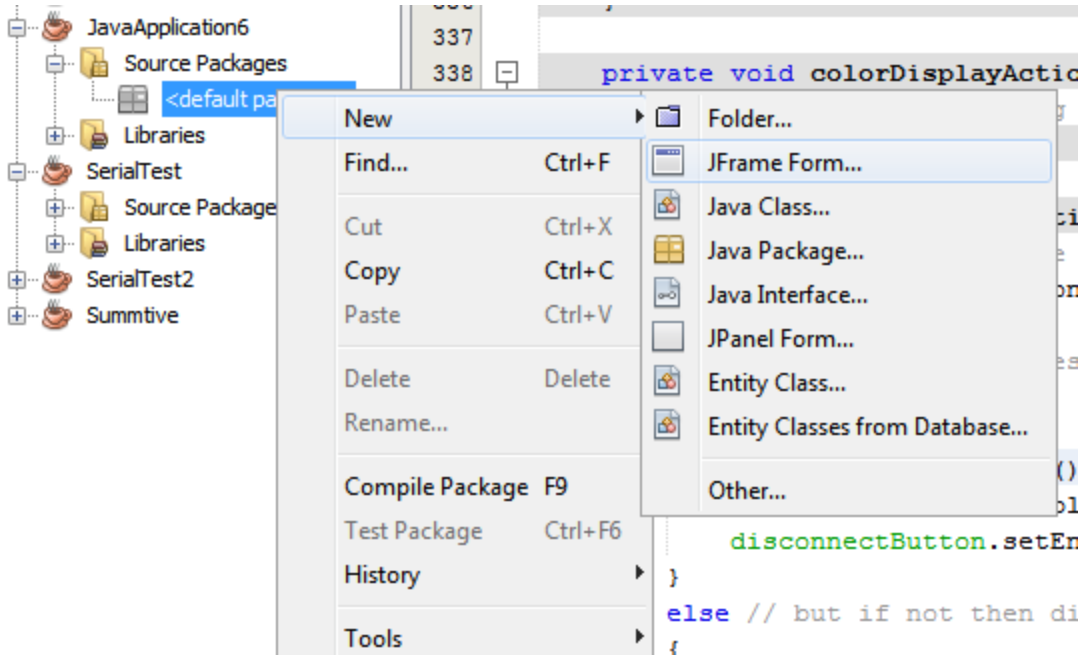
2. Click on the Folder that says Java and on the right click on the first option that says Java Application.

3. On the next window specify what you want to call your project and where you want your project to be saved to. Then leave the Create Main Class box checked but get rid of what is in the field and click finish. (Shown below)



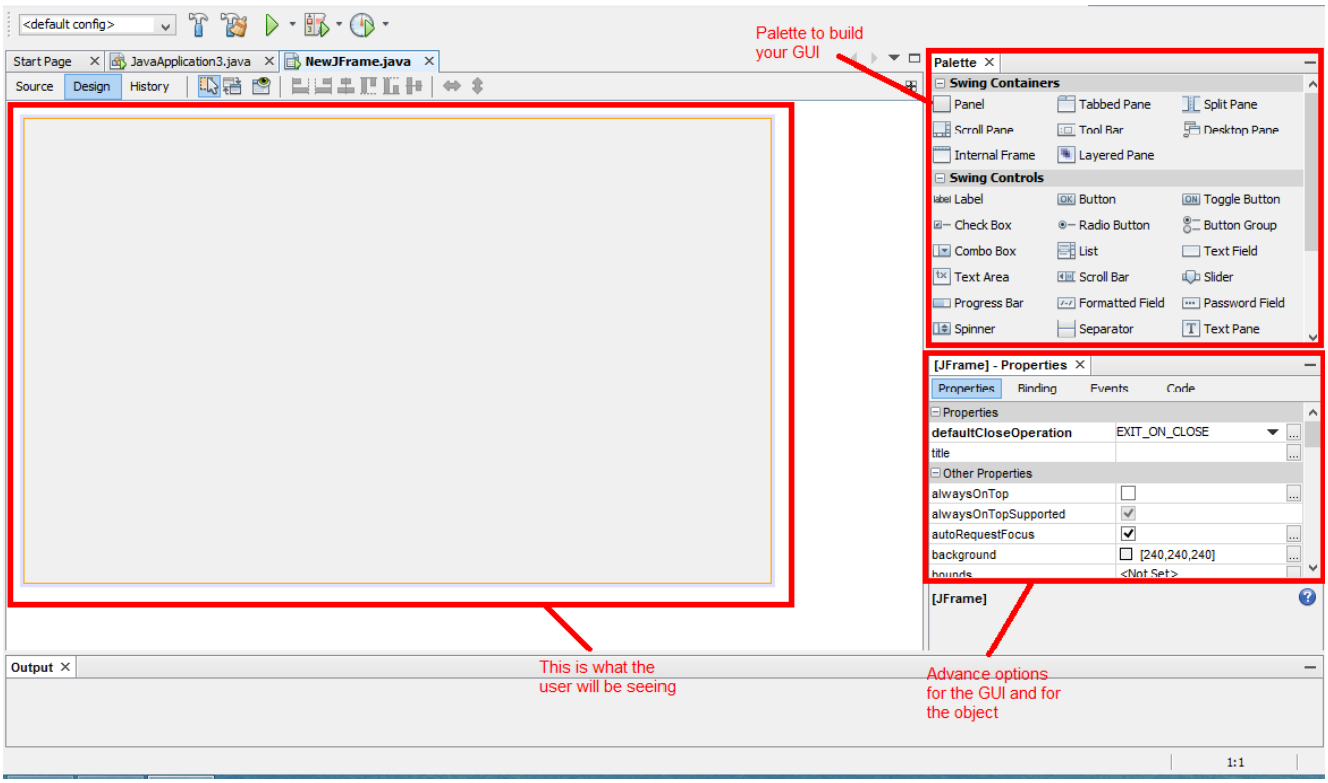
4. On the left in the project pane you should see the name of your project. Press the + to reveal a folder called source packages if you cannot already see it. Right click on Source Packages and highlight New and then click Java Package. On the window that shows up, specify the name of the package on in the text field on top of the window and then click finish.

5. Now right click on the new package and then highlight New and then click on JFrame Form. (Shown below - similar of that of the last instruction)



6. On the window that appears give your class a name in the Class Name box and then click finish. You should see the empty layout of the application with buttons on the right.

Let's introduce you to the layout. The grey box in front of your main panel is called the "JFrame". This panel is what the user is going to see when we run the program. Also, it is where you will be putting all of your layouts on. On the right side you will see a panel called "Palette" those are all the tools you need to build your GUI. Finally below that is the properties menu. This is where you can adjust more advanced settings for the things you created. **Use the next image for reference.**



Now we will familiarize ourselves with the Java GUI palette. To build our basic GUI all we need is 17 labels, 13 text fields, 2 buttons and 1 text area. All of these objects can be found in the palette→Swing Controls on the right hand side.

Let's explain what the components we will be using for the GUI are before we get into using them.

JButtons – These are buttons that the user can press to trigger and event such as displaying text, saving files, doing arithmetic and displaying text, closing the program etc. What we will be doing in our project is to display data on the text boxes read from the Arduino. There are also toggle buttons, radio buttons, check boxes but we will use the regular button.

JTextField – This is used either for getting input from the user or displaying data. Textboxes can be editable for when we want to get user input or they cannot be editable if we only want to display data. We will be using them to display data.

JLabel - A JLabel is nothing more than the name implies. It is for labeling text boxes, displaying titles, or pretty much displaying whatever text on the screen that you like. You can change the text by simply double clicking it.

JTextArea – JTextArea is very similar to a textbox but this is a large textbox that allows the user to see large amount of text that they can scroll through. We will only use this as a console window to output anything related to connecting or disconnecting for error and exception handling.

Placing the basic components:

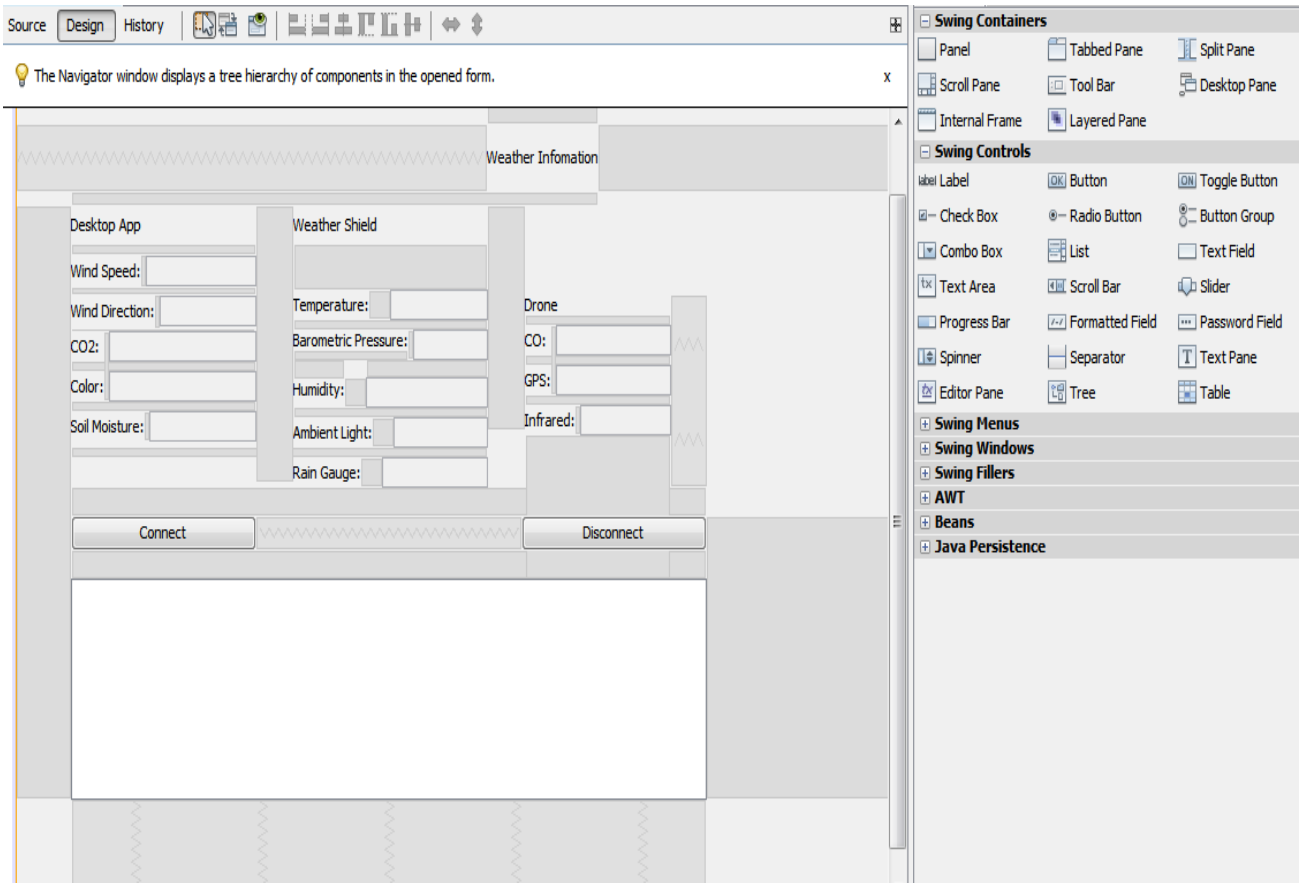
JLabel:
Drag and drop a label from the Swing Controls onto the JFrame near the top. Double click the label and change its name to "Weather Information" without the quotes. We have now put a JLabel on the JFrame.

JButton:
Now let's pick up a 'button' and place it on the frame. Right click on the button and click Edit Text. Name the button "Connect." Place another button and name it "Disconnect."

JTextField:
Now take a text field box and place it on the program. Right click on it and click Edit Text. Just backspace all of the characters to make sure that it is blank and press enter. Note the text field adjusts according what is in the fields. You can click on the text field and then click on one of the squares around the box and drag it to make it bigger.

JTextArea:
Take a text area from the right and place it near the bottom of the frame. Extend it to the right a bit.

TASK: Add and move components around to make the GUI look similar to this:

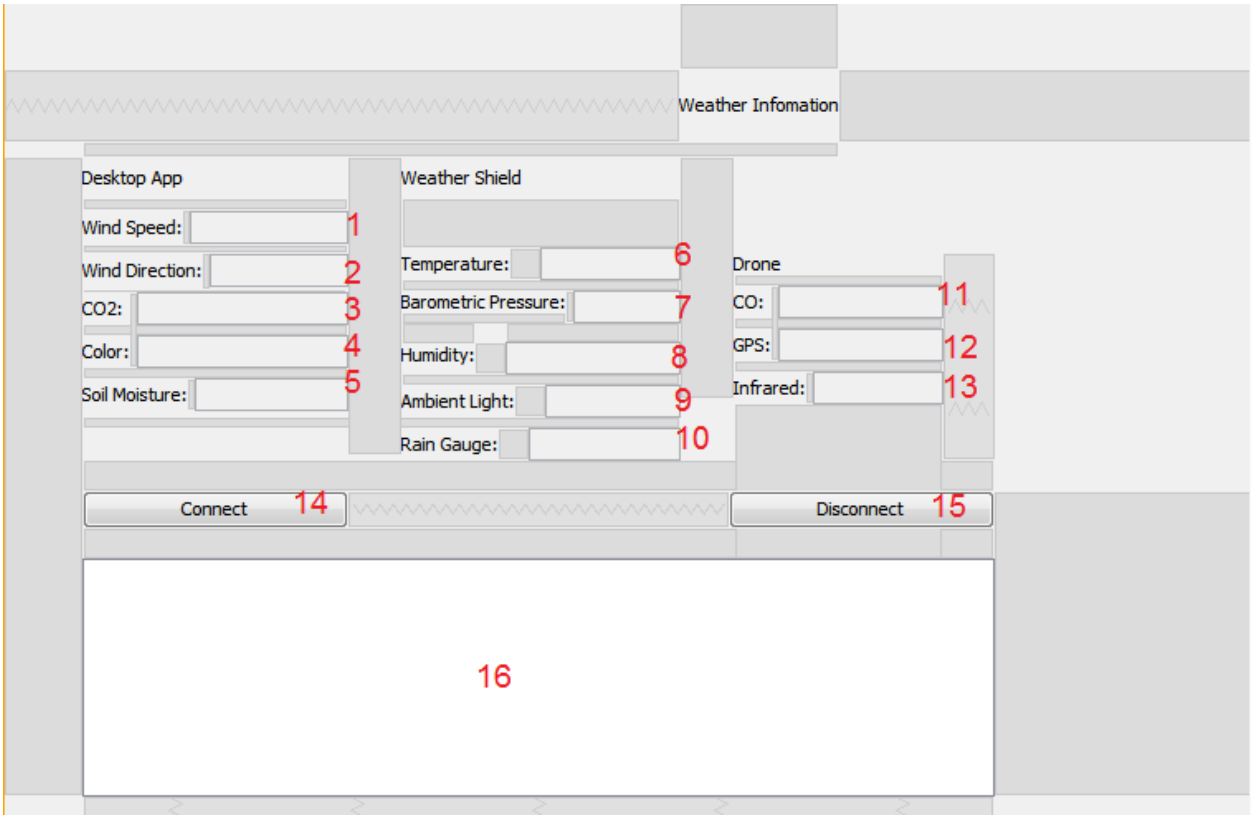


Note: Since you already placed the buttons and the text area all you need to do is place labels and text fields. Use the instructions above for reference. Your GUI should look like this.

Variables and Properties

Now we have placed text fields, buttons, labels, and a big text field area we need to configure the variable names and the properties for the components for our GUI.
To set a variable name simply right click on a component, click on Change Variable Name and change it to what you prefer. The GUI builder will give the components names by default but it is recommended to give then meaningful name so you can program and debug easier! For our program we can leave the label name as their default names but we need to give the buttons and the text boxes names.

TASK: Change the variable names to the ones specified below. Note: You must give them the variable names below to prevent confusion and errors later on in this manual as we will be giving code to get the data acquisition working later.



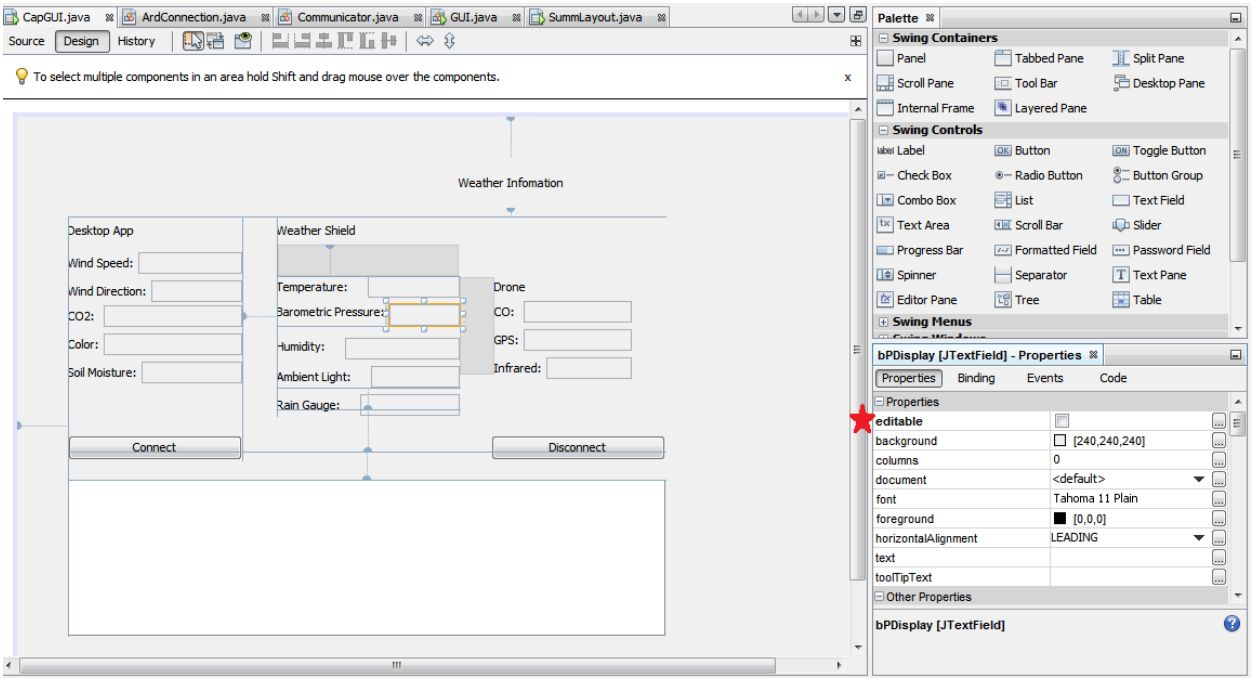
1. **windSDisplay**
2. **windDDisplay**
3. **coTwoDisplay**
4. **colorDisplay**
5. **soilMoistDisplay**
6. **tempDisplay**
7. **bPDisplay**
8. **humidityDisplay**
9. **ambientLDisplay**
10. **rainGDisplay**
11. **coDisplay**
12. **gpsDisplay**
13. **ifrDisplay**
14. **connectButton**
15. **disconnectButton**
16. **consoleBox**

Now before we get to the data acquisition we need to customize two more things, changing the textboxes to non-editable and setting them to public. We don't want the values in the text boxes to be changed by the user and we want the program that will be given to be able to manipulate the textboxes directly in order for the data to be displayed. We also want the disconnect button disabled on default. Our program will turn that on when we connect to the board later on.

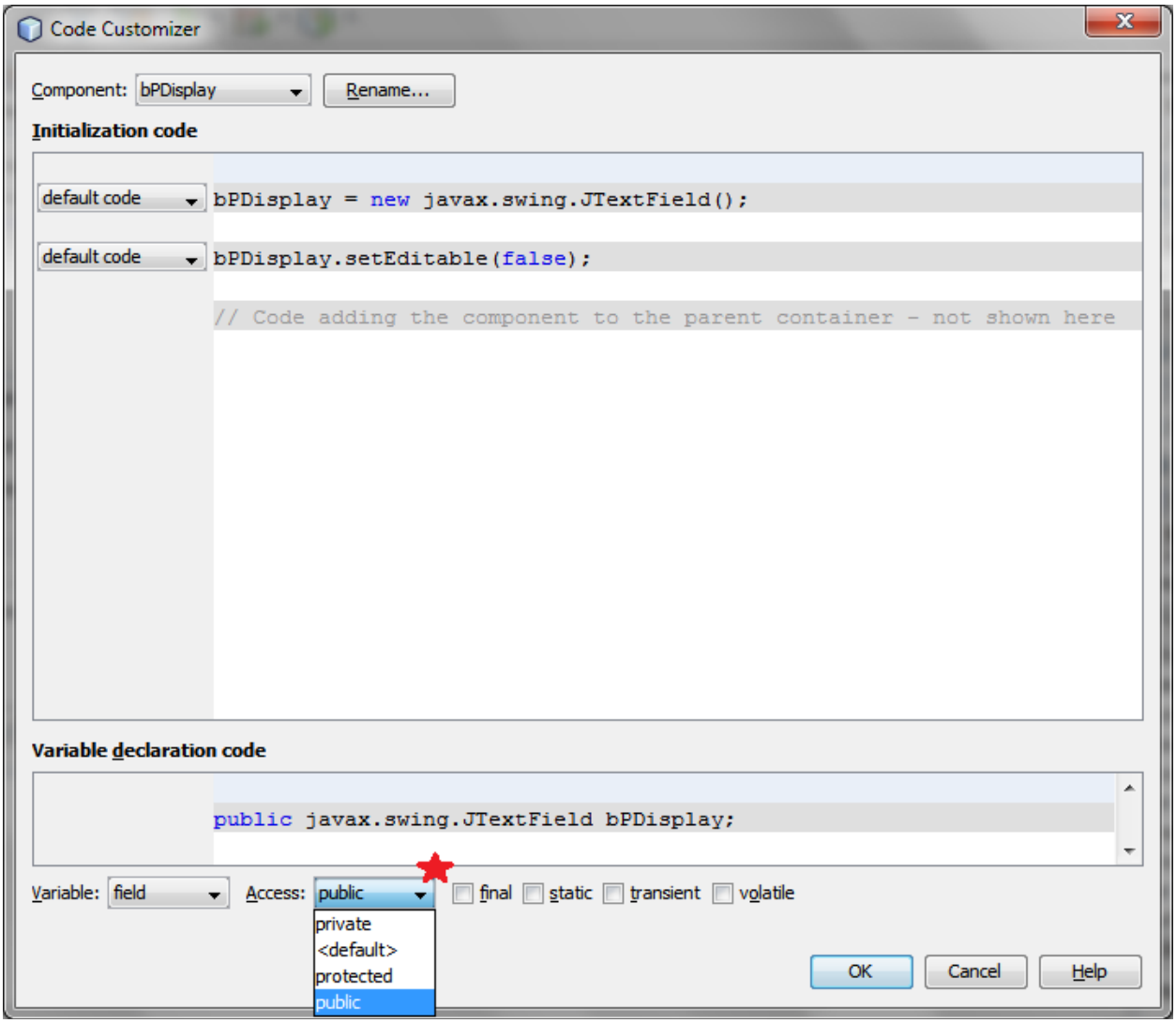
So let's do an example:

For the disconnect button left click it and on the properties window on the right scroll down and look for a property that says enabled and uncheck that box or right click the button and then click properties and look for the enabled property and uncheck.

Now let's do that to the text boxes. Left click on any textbox and in the right window uncheck editable. Alternatively right click on the textbox and click properties. Near the top of the window uncheck editable.



Now right click the text box and click Customize Code. At the bottom of the window click on the drop box and then click on public.



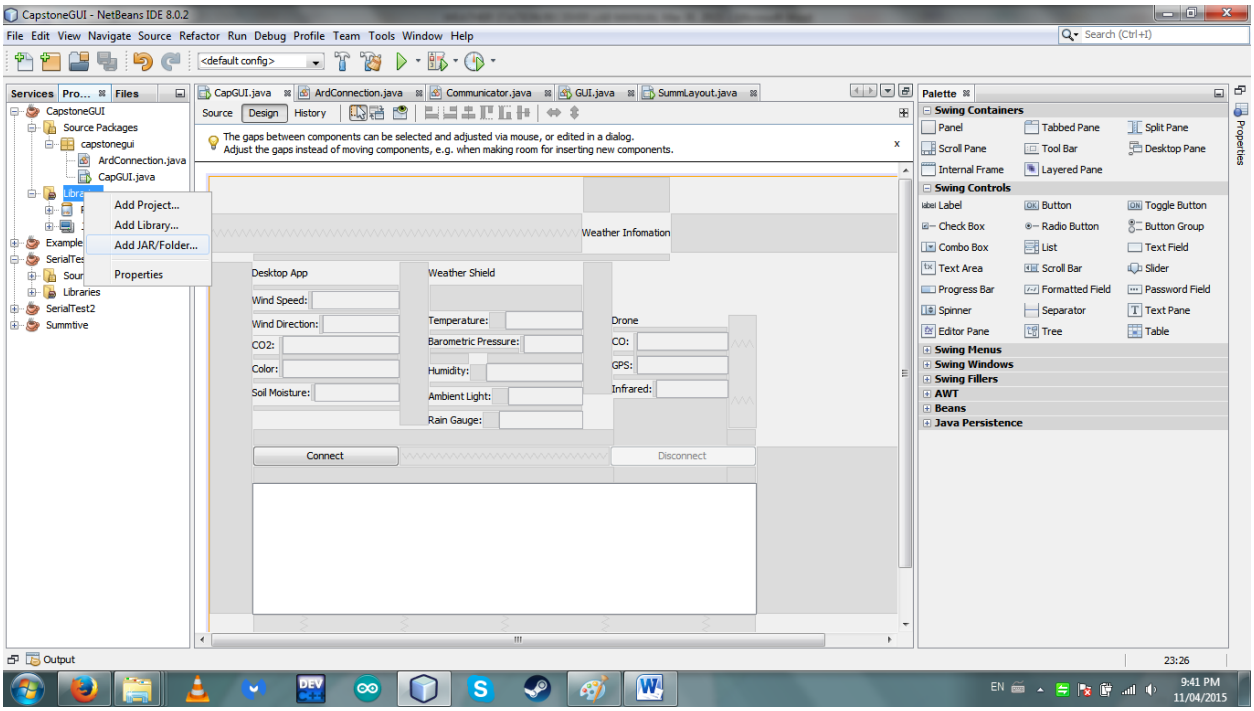
TASK: Do this for all of the text boxes and the text area too.

Installing RXTX library for the GUI

Now that we are finished building the GUI let's move on to installing RXTX. RXTX is used in Java programs to communicate with serial devices. We will be installing the Ardulink library instead of manually installing RXTX because that contains both 32 bit and 64 bit RXTX files that can be installed and there is also a batch file there that will take care of the rest of the installation. Note: You cannot run a 32 bit RXTX in a 64 bit Netbeans and so on. It is important that you install the right one for the Netbeans and the Java IDE you installed. If you are not sure what version you installed, try installing both but be warned. Click here to download Ardulink.

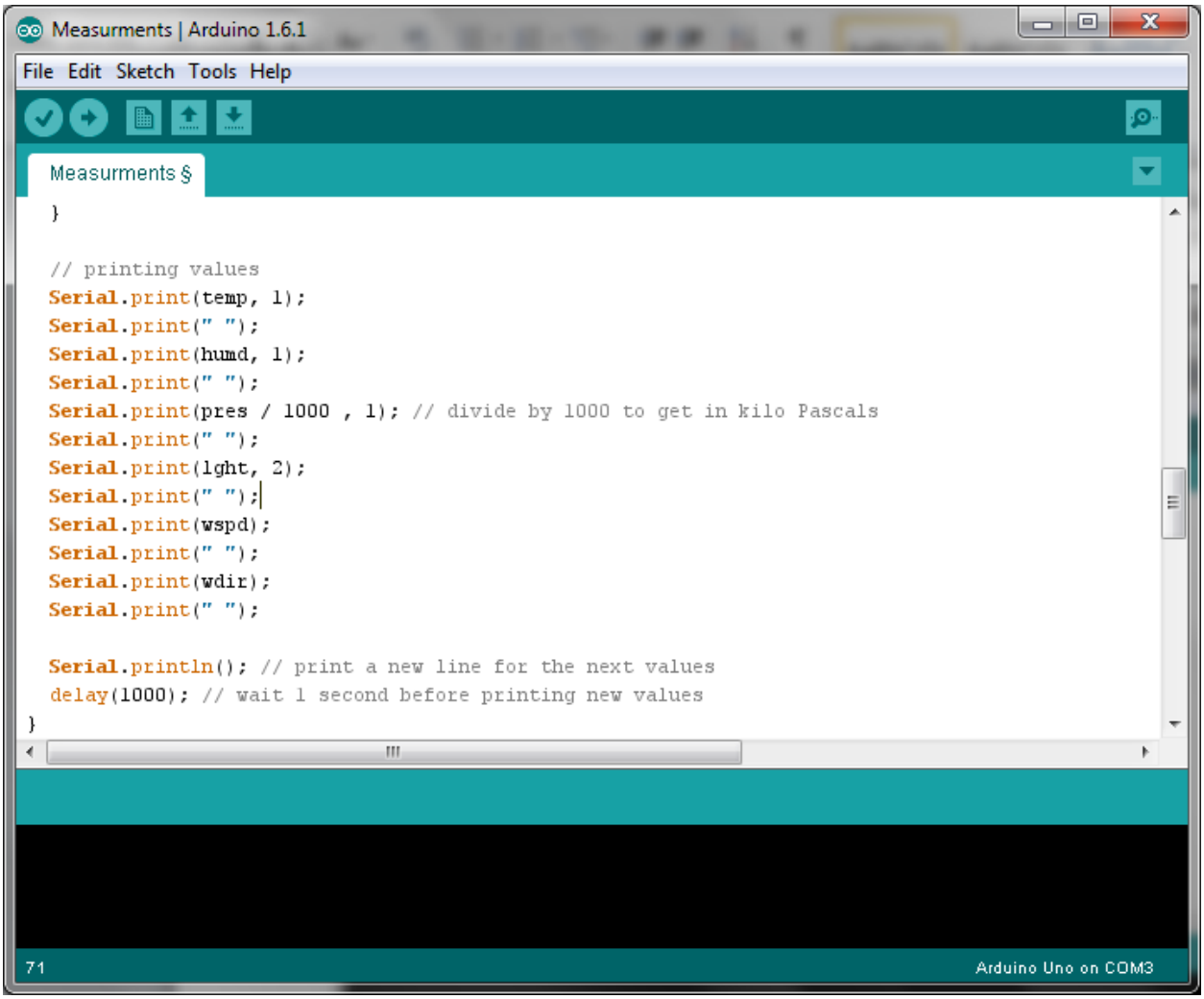
When you completed the download of Ardulink extract the contents of the zip file anywhere you like and then locate and open the folder. Near the bottom of the list of files there should be two files, set32bitWindowsRXTX and set64bitWindowsRXTX. Open the version that is the same as the one you downloaded the Java IDE and Netbeans. (Ex. If you downloaded 64 bit IDE and Neatbeans, install the 64 bit one).

Now let's import the library into our java program. On the left where the project is on Netbeans right click the libraries folder and click Add JAR/Folder.

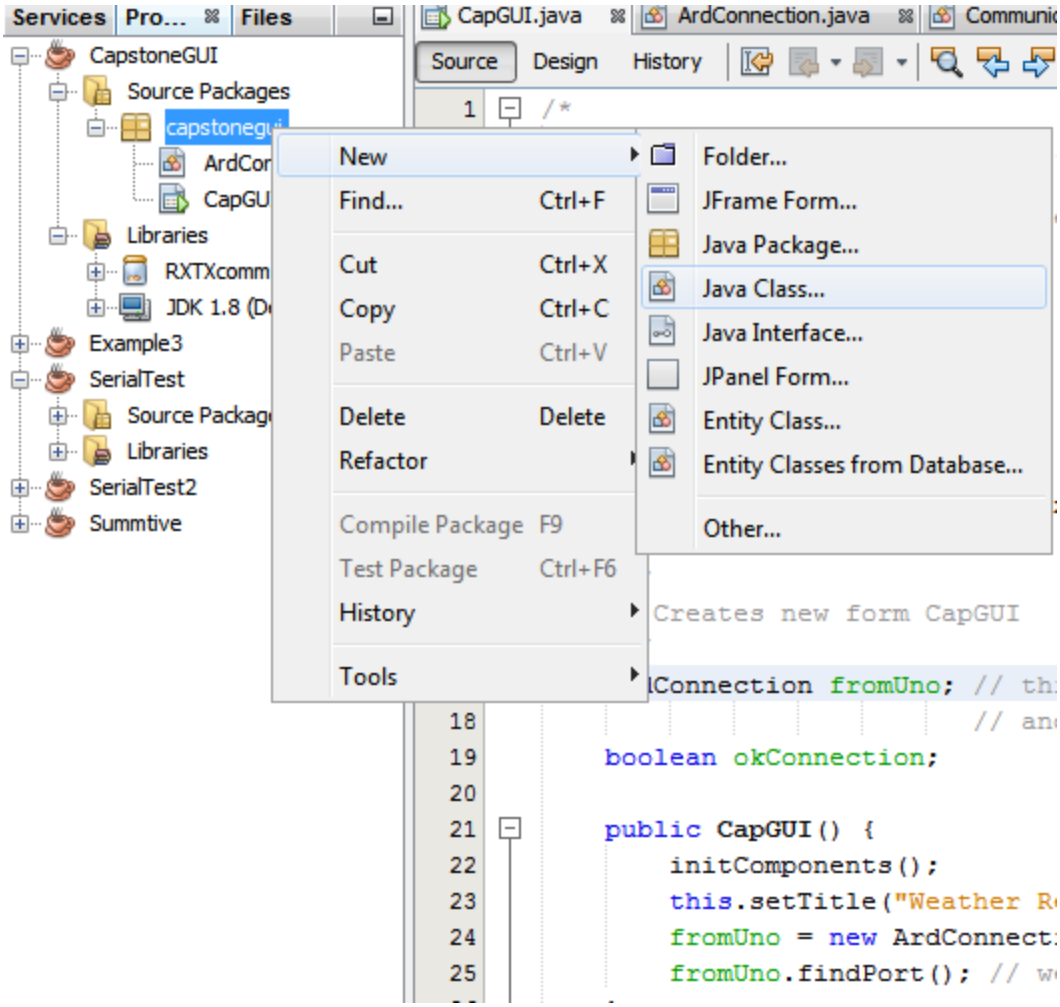


Locate the ardulink folder you have extracted. In the ardulink folder there are two folders called 32bit and 64bit each with a file called RXTXcomm. Open the folder of the RXTX version you installed and open the jar file that is in there. So if you installed 64 bit RXTX then open the 64bit folder and open the jar file that is in there.

Now let's do one more thing with the Arduino code we typed earlier. Open the code that we typed that reads the temperature, humidity, barometric pressure, ambient light, wind speed and wind direction. Let's clean the code so that it only prints the value and a space after the value. This will make the program easier to read the numbers as it will be the only thing the Arduino will send out! Also our program might encounter errors if anything else is sent to the program. This is what the Arduino code should now look like:



Now it's time for the sample code part. On the left of the Netbeans program right click the package you clicked on earlier and highlight New and then click Create Java Class.



In the class name box name your class ArdConnection to prevent any errors when you copy the code.

Now in the new class file you created press ctrl+a and then press backspace. Copy and paste this code:

Note: This code may be difficult to highlight and copy. Try to highlight and copy and paste one page at a time and paste into the IDE at exactly where you left off to avoid errors. There should not be any formatting errors when you paste into the IDE but it may happen. Make sure the blinking cursor is at the far upper left of the IDE when you paste.

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this datalate file, choose Tools | Templates
 * and open the datalate in the editor.
```

```
 */
package capstonegui;
import gnu.io.*;
import java.io.IOException;
import java.io.InputStream;
import java.util.Enumeration;
import java.util.HashMap;
import java.util.TooManyListenersException;
/**
 *
 * @author Alexander Blake
 */
public class ArdConnection implements SerialPortEventListener {
    // declaring and initializing variable to use for the connection
    CapGUI gui = null;
    private Enumeration port = null;
    private CommPortIdentifier portID = null;
    private SerialPort serial = null; // serial port with the uno on it
    private InputStream sInput = null; // get output from arduino/input to pc
    private HashMap hashPort = new HashMap();
    private String foundPort;
    private final static int TIMEOUT = 1000; // the read rate

    // declaring and initializing variables for reading the data fom Uno
    private String data = ""; //
    private int pointer = 1; // variable that tells the program what text box to print to
    private String gpsCords = ""; // variale so we can store the entire gpd coordinates in one
string

    // this constructor allows us to use swing components from the main class
    // that are set to public so we can manipulate them here in this class
    public ArdConnection(CapGUI gui)
    {
        this.gui = gui;
    }
```

```
   // this method searches all the com ports to find the arduino board to connect to
   // the com port it is on will vary between operating systems so this is needed
   public void findPort()
   {
      port = CommPortIdentifier.getPortIdentifiers(); // initializing value

      // this searches all of the ports and find what com port the arduino is connected to
      // the arduino is connected to a serial port (USB) so we search until we find a
connection
      while (port.hasMoreElements())
      {
         // get the port that the program is looking at
         CommPortIdentifier selectedPort = (CommPortIdentifier)port.nextElement();

         /* if the port that it is looking at is a serial port
          * get the name of the port so we can connect to it later on
          * and put the port name and what it is in the hash map also for connecting
purposes */
         if (selectedPort.getPortType() == CommPortIdentifier.PORT_SERIAL)
         {
            foundPort = selectedPort.getName(); // copy its name into the string
            hashPort.put(selectedPort.getName(), selectedPort); // put the port in the hash
map variable
            gui.consoleBox.append("Connection on Port: "+selectedPort.getName()+"\n");
         }
      }
   }

   // this method establishes a connection when called
   /* note: thee findPort() method must be called first so the program know what port it
should connect to! */
   public boolean connectToBoard()
   {
      boolean connectionOK = false; // verifying connection
      // this gets the id of the port we need to connect to
      // we need to get the found port and convert it to a commportidentifier.
```

```
        // this is why we needed that string variable and hash map variable
        portID = (CommPortIdentifier)hashPort.get(foundPort);

        CommPort cPort; // variable that will have the open port with the uno on it

        try
        {
           // open the port that has the id we needed to connect to
           cPort = portID.open("ArduinoUNO", TIMEOUT);
           serial = (SerialPort)cPort; // and convert that
           connectionOK = true;
           gui.consoleBox.append("Connection successful. "
                 + "Disconnect when finished.\n");
        }
        catch(PortInUseException e){ // exception if this port was already in use
           gui.consoleBox.append("Error: This port is in use.\n");
        }
        catch (NullPointerException e) // exception if user forgets to connect board or other
connect error
        {
           gui.consoleBox.append("Connection Failed. Arduino may not be connected to
port.\n"
                 + "Verify connection and restart program.\n");
        }
        return connectionOK;
     }

     // this will start reading the data
     // note: that this will also call the serialEvent method at the bottom
     public void startReading()
     {
        try {
           sInput = serial.getInputStream(); // get the input from the uno
           serial.addEventListener(this); // we need to listen for when the uno sends data
           // the event listener will tell the program there is data available only when it
receives it
```

```
            serial.notifyOnDataAvailable(true); // and then parse the data in the serialEvent
method

        } catch (IOException ex){
            gui.consoleBox.append("IO Error.\n");
        }
        catch (TooManyListenersException e){
            gui.consoleBox.append("Too many listeners.\n");
        }
    }

    // this method will disconnect from the port in order to prevent errors
    public void disconnectFromBoard()
    {
        try{
            serial.removeEventListener();
            serial.close();
            sInput.close();
            gui.consoleBox.append("Discconnected successfully.\n");
        } catch (IOException ex) {
            gui.consoleBox.append("IO Error\n");
        }
    }

    // this method parses the data when the data is received
    // it will get the data into individual variables so
    public void serialEvent(SerialPortEvent evt)
    {
        // when data is available we will parse it and store their values in seperate variables
        if (evt.getEventType() == SerialPortEvent.DATA_AVAILABLE)
        {
            try
            {
                int rawData = sInput.read() - 48; // read the data unconvert from its ascii
equivalent
```

```
/* when are deconverting from ascii code the goal is to get the value
 * to the number we want as we don't need to show its ascii codes
 * ex: ascii for 2 is 50 but we don't want 50, we want 2 so that is why
 * we are subtracting 48. however non-digits such as decimal places
 * show up as negative values and we need to do special operations on them.
 */

if (rawData == -2) // if the data is a decimal put a decimal dot on the string
{
   data = data.concat(".");
}
else if (rawData == -3) // if the data is a minus sign, put one on the string
{
   data = data.concat("-");
}
// if the data is a space we need to check the pointer to see
// what text box we need to print the data to.
// note: arduino is going to send specific data so we need to know
// the right place to put it. temp. data to tempbox. humid. to humid. etc
else if (rawData == -16)
{
   // check pointer so we can print data to respective text box
   if (pointer == 1) // print to temperature
   {
      gui.tempDisplay.setText(data + " °C");
   }
   else if (pointer == 2) // print to humidity
   {
      gui.humidityDisplay.setText(data + " %");
   }
   else if (pointer == 3) // print to pressure
   {
      gui.bPDisplay.setText(data + " kPa");
   }
   else if (pointer == 4) // print to light
   {
```

```
                gui.ambientLDisplay.setText(data);
            }
            else if (pointer == 5) // store gps latitude in coordinates string
            {
              gpsCords = (data + ", ");
            }
            else if (pointer == 6)
            {
                gpsCords = gpsCords.concat(data); // add gps longitude to coordinates
string
                gui.gpsDisplay.setText(gpsCords); // place coordinates string to text box
            }
            else if (pointer == 7)
            {
                gui.windSDisplay.setText(data + " km/h"); // print to wind speed
            }
            else
            {
                String degWord = getWindDirWords(data); // get the wind direction in
words
                gui.windDDisplay.setText(data+" degrees ("+degWord+")"); // print wind
direction degrees and word
                /* so now that all the data is printed we want to reset the
                 * pointer buuuut every time we write data the pointer increments
                 * we want the pointer to now point to 1 but since we just printed
                 * data it will point to 2 and mess up the printing order
                 * soooo for compensation purposes we reset it to 0 so when it
                 * does increment it will increment it to 1 so we can keep printing
                 * in the order we want */
                pointer = 0;
            }
            // every time data is printed, clear the data string we built
            // so we can write the next value and increment the pointer so
            // we can print to the next proper text box
            data = "";
            pointer++;
```

```
            }
            /* when we reach the end of the data line the arduino sends out to java
             * we want to make sure it does not attach garbage to the data string. */
            /* because this only happens when all of the values are parsed
             * we want to keep the data string empty */
            else if (rawData == -35 || rawData == -38) // i noticed -35 and -38 is the
garbage it sends after a line
            {
               data = "";
            }
            // if the data is not what we were trying to filter (ie non-digits)
            // attach the data to the data string to build out the digit to print out
            else
            {
               data = data.concat(Integer.toString(rawData)); // put that string into
another and build on it by concat()
            }

         }
         catch (IOException ioe)
         {
            gui.consoleBox.append("IO Error\n");
         }
      }
   }

   // get the wind direction in words based on what the degrees are
   private String getWindDirWords(String deg)
   {
      int degrees = Integer.parseInt(deg);
      if (degrees == 0) return "N"; // north
      if (degrees == 45) return "NE"; // north-east
      if (degrees == 90) return "E"; // east
      if (degrees == 135) return "SE"; // south-east
      if (degrees == 180) return "S"; // south
      if (degrees == 225) return "SW"; // south-west
```

```
        if (degrees == 270) return "W"; // west
        if (degrees == 315) return "NW"; // north-west
        return "Error"; // in case of non maatch
    }
}
```

When you paste the program there will be some errors but don't worry about any of them as these should only be naming issues we will fix right now. Let's change these lines: You can find them all near the top of the code you copied.

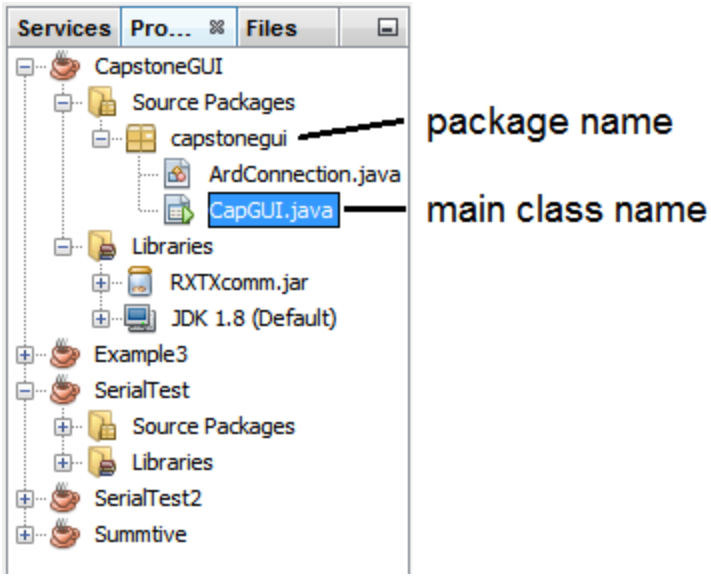
```
package capstonegui;
CapGUI gui = null;
public test(CapGUI gui)
```

Now here is where you will get errors that are easily fixable.
First change this line to the name of the package you created when you started a new project in Netbeans.

Before:
```
package capstonegui;
```

After:
```
package /*name of your package*/
```

You can find the name of your package here:



Now the next two lines:
CapGUI gui = null;
public test(CapGUI gui)

Simply change the CapGUI part to the name of your main class you created. See the above picture for an example.

Before:
CapGUI gui = null;
public test(CapGUI gui)

After:
/*name of your main class*/ gui = null;
public test(/*name of your main class*/ gui)

Once you do this save your file. At this point all other errors should vanish if you named the variables on the gui properly and installed the RXTX library.

Now let's briefly take a look into to the program to see how it works.

Of course at the beginning we are importing classes from the libraries we are using just like we did in the Arduino program.

Notice how we are implementing a class called SerialPortEventListener.
This is an abstract class. We must use a method called serialEvent in order to use it properly as that is the only method in that class. Note when you implement an abstract class you must use all of the methods inside the abstract class or you will get compile errors. Now what this is doing is listening on the port for any data that gets sent. It will only trigger when there is data available therefore saving CPU overhead.

Next part is the findPort method. Note this was created and not needed but remember it's good to create methods when you want your program to do specific things over again and it really helps when you want your program to do a specific function. Just create a method and call it. Anyway what we are doing here is finding what port the Arduino is connected to so we can connect to it when we press the connect button later on.

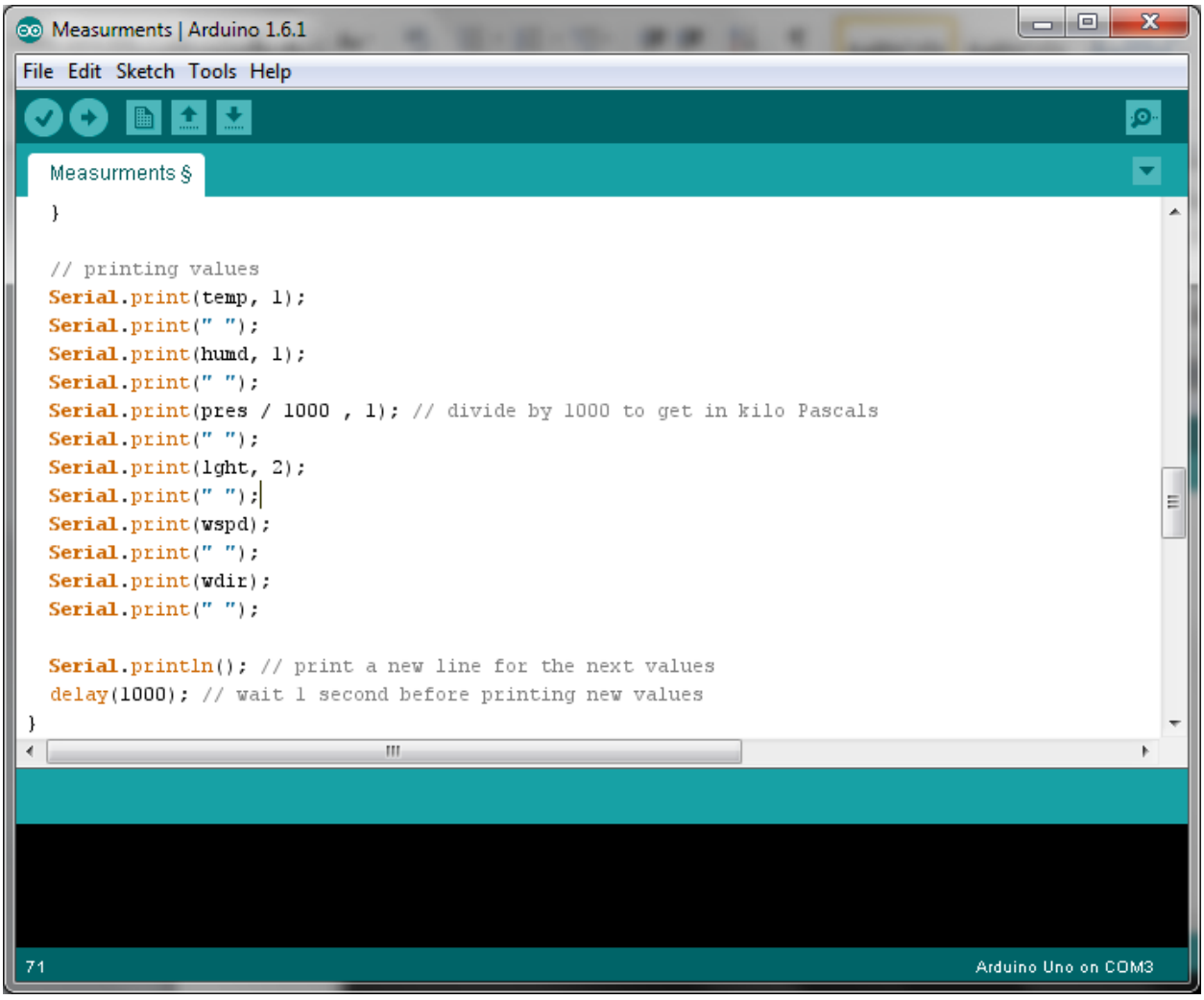
Next is the connectToBoard method. It's self-explanatory. What this is doing is getting the port it found the Arduino earlier and opens it and connects to it.

Next is the startReading method. This gets the input from the Arduino and also creates the listeners so it will only process the input when it actually receives it.

Next is the disconnectFromBoard method. Again it's self-explanatory. This will disconnect from the board and shut down the listeners. It's important to do this before you close the program in order to prevent possible errors with the port being in use when the program is closed. It may not be necessary to do this Windows but just in case!

Finally the last method is the abstract method we needed to create. Remember how we talked about interrupts last week? This does it. Every time there is data on the serial, the action listeners created in the connectToBoard method cause an interrupt. The program then executes the code in this method. This method reads the raw data and puts it into a string. It will read one character at a time and each character will be added onto the end of the string until a space is found. Then the string will print to a certain text box depending on what value the pointer integer variable is at when the space came in until it gets a new line character where then the pointer is reset and the process repeats. This is why it was important that we modify the Arduino code.

Note: The program will assume at this point temperature is the first value coming in and then humidity, pressure, and then light. Make sure the part of your Arduino program responsible for printing looks exactly like this!



We will add on to the Arduino code eventually. Forget about the lines that print the GPS coordinates for now. We will cover the GPS in depth in week 7.

This was only a brief look at what the program is doing. **For more information, refer to the comments in the code.**

Now let's get out program buttons working and ready. First go back to your **main program**. Look at the top of your program and you should see this:

public /*name of your main class*/ {
initcomponents();

This is the constructor for your program. Before the constructor in the space above we would want to declare an instance of our ArdConnection class and a Boolean for disabling the buttons when connected or disconnected.

Copy and paste:
ArdConnection fromUno;
boolean okConnection;

In the constructor we can set up the window any way we like and also we can do anything we want the program to do on start up. Let initialize the fromUno variable, start looking for ports and give the program a name.

Copy and paste into constructor:
fromUno = new ArdConnection(this); // initailize
fromUno.findPort(); // we need to find what COM port the Uno is on

You don't need to copy comments but remember it's good practice to make comments! Now let's give the program a name. All we need to do is put this.setTitle(""); in the constructor and inside the quotes put whatever name you like. **Now try it!**

When you are done the constructor should look like this:

```
ArdConnection fromUno;
boolean okConnection;

public CapGUI() {
    initComponents();
    this.setTitle("Weather Reader");
    fromUno = new ArdConnection(this); // initailize
    fromUno.findPort(); // we need to find what COM port the Uno is on
}
```

We are almost done. Now let's get the buttons working. Switch back to the Design tab and double click on the button that says connect. What is does is add a listener that checks if the button was pressed and in that part of listener where the comment says todo code here we make our program connect to the board and if that was successful it will turn on the disconnect button, turn off the connect button, and start reading data.
Here is the code for that.

```
// we will connect to the arduino and then be able to read data after
        okConnection = fromUno.connectToBoard();

        // if connection is successful startreading and display
        if (okConnection)
        {
            fromUno.startReading(); // start reading data
            connectButton.setEnabled(false); // we don't need to click this button again once
connected
            disconnectButton.setEnabled(true); // we can disconnect now that we have a
connection
        }
        else // but if not then disable all the buttons and don't read
        {
            connectButton.setEnabled(false);
            disconnectButton.setEnabled(false);
        }
```

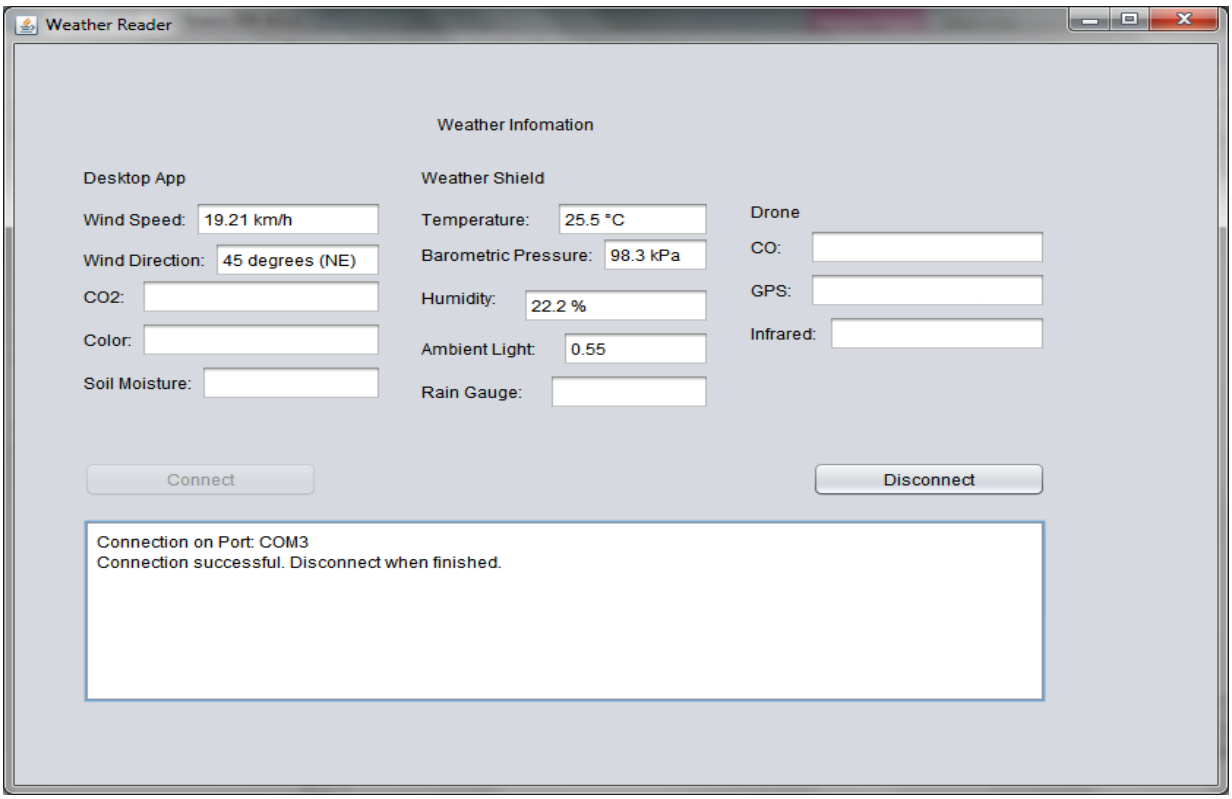
Now the disconnect button. It's very similar the connect button but less code. Go to the gui and then double click the disconnect button.

Copy and paste:

```
// we are going to disconnect from the board
        fromUno.disconnectFromBoard();
        connectButton.setEnabled(true); // we can reconnect if we want to
        disconnectButton.setEnabled(false); // we disconnected so we don't need to press
the button again
```

Now you got it all there. If you followed the instructions for the main program part, you should not have any errors, especially if you follow the instructions where we name the variables on the GUI properly! You should save your program and connect your Arduino board now.

Once you connect you Arduino and run your program you should see a message in the text area that confirms that the Arduino was found. Click connect and watch the data go into the text boxes! Click disconnect when finished before closing the program. The output should look similar to this:



One last thing: there was a line of code near the beginning that looked like this:
final static int TIMEOUT = 1000; // the read rate

As you may guess this is the delay between reading the port. It is set to delay at 1000 ms – the same as your aduino program at this point. Try changing this but make sure to set it back to 1000.

Next we will save this data it in a graph sheet – the .xml file and later try to make graphs.

WEEK 5

Objectives for this week:

- Logging and displaying weather data.
- Understanding Dashboards

WEEK 6

Objectives for this week:

- Introduction to XBee wireless transmission.
- Remoting your weather station

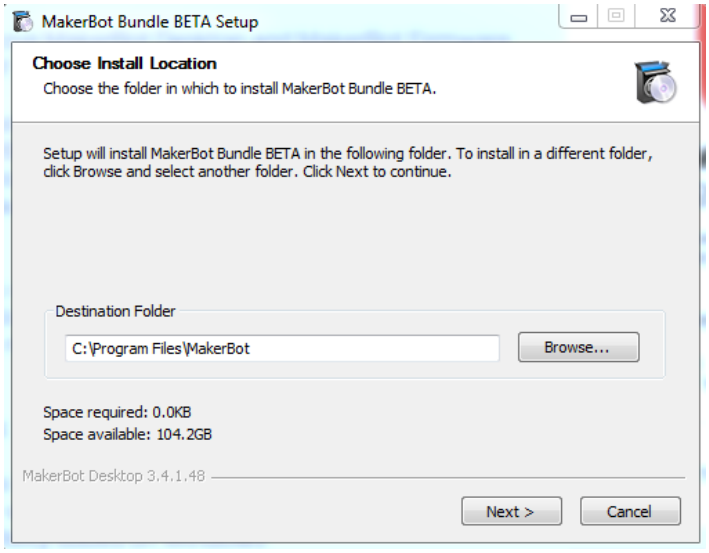
WEEK 7

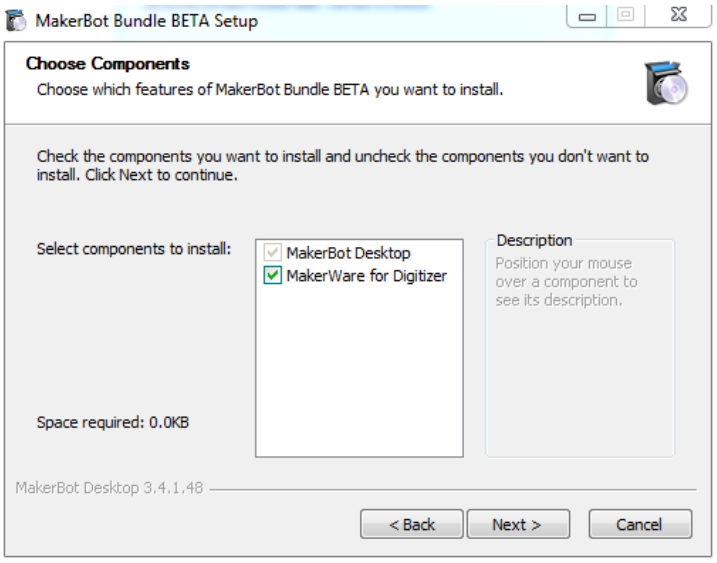
Objectives for this week:

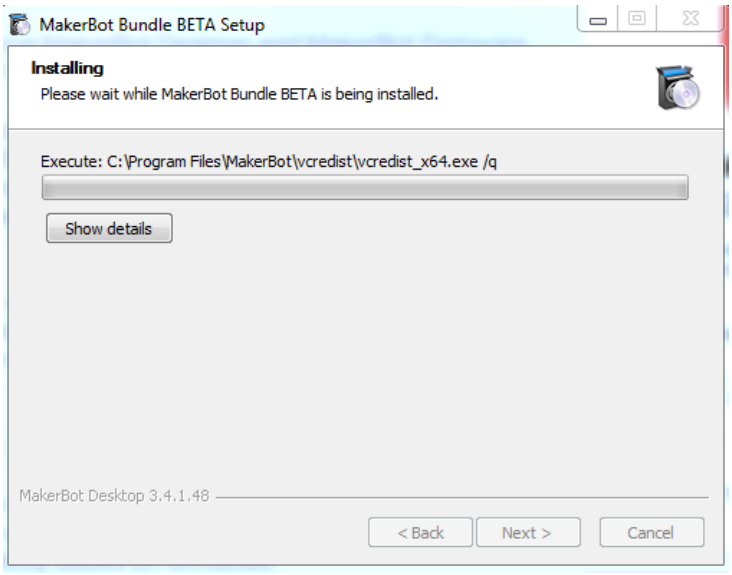
- Adding global Positioning System (GPS) information to your weather shield.
- Introduction to 3D printing software concepts
- Using 3D printing (SolidWorks or TinkerCAD and MakerBot) software to make your own anemometer

MakerBot

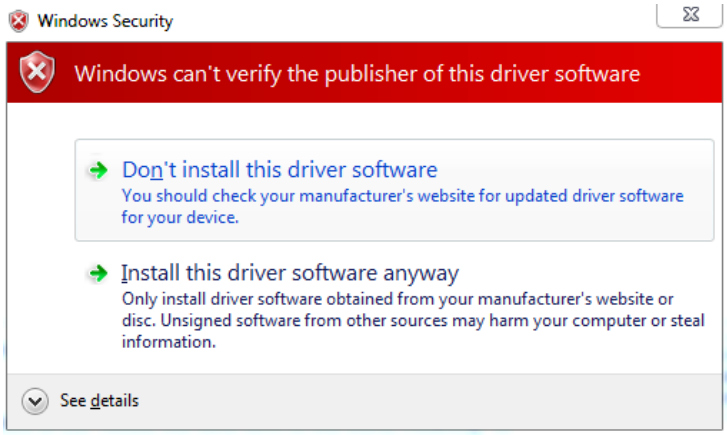
Download and install MakerBot Desktop software http://www.makerbot.com/desktop (version we are using is 3.4.1)

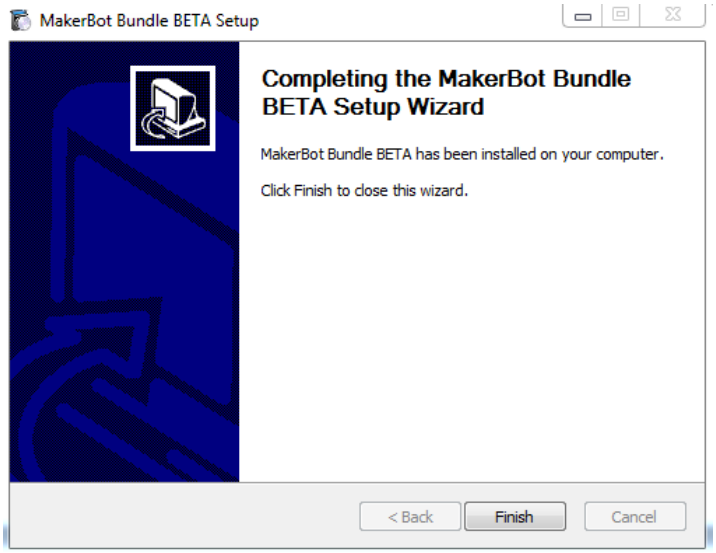




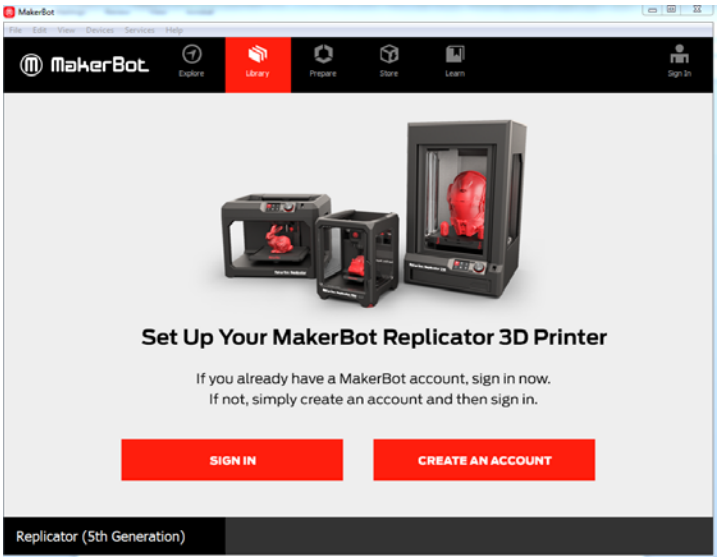


You may see some Windows Security warnings while installing the software

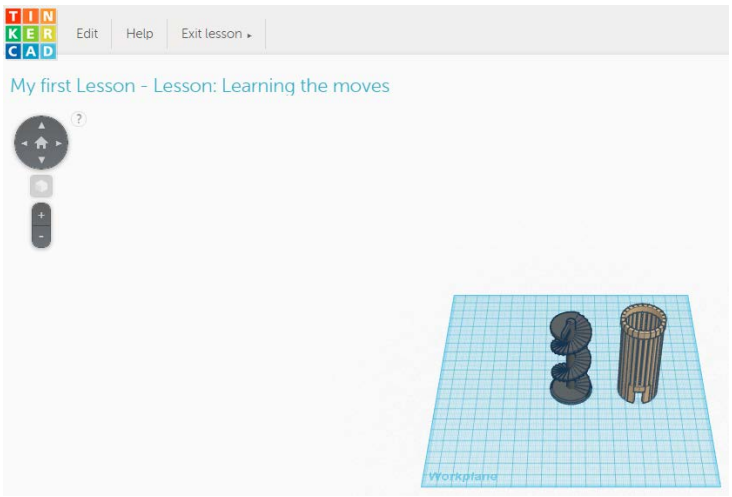




- Watch the videos in the introduction screen such as https://www.youtube.com/watch?v=5JwwLizuWRY&feature=youtu.be
- Create a MakerBot account



- Create an account with Autodesk's TinkerCAD 3D software application https://www.tinkercad.com/



Adding GPS information to our weather shield

Now that we know how to use the maker bot let us focus on the GPS part. What we need for this is the TinyGPS++ libraries (here is the link to the website to download it: http://arduiniana.org/libraries/tinygpsplus/). Our main goal here is to get GPS to communicate back to us and give us the longitude and latitude of our location. The first thing we want to do is to import the libraries of the GPS. Now that we done that, open up your ardunio program on your computer and we add the header file at the top of the program for the tiny GPS and the softserial:

#include <TinyGPS++.h,
#include <SoftwareSerial.h>

(*Note* the softserial is an interface that allows the GPS to communicate with the weather shield). We now need to declares an instance of the TinyGPSPlus object by adding

TinyGPSPlus gpsplus;

Last but not least, we declare an instance of the softserial as well as writing the communication ports by writing

SoftwareSerial softserial(5, 4);

(*Note* the 5 and 4 are the pin numbers for the weather shield; 5 is the receiving pin while the 4 is the transmission pin).
Next step is to get the GPS to start by adding this line "softserial.begin(9600);" in the void setup() method; this line gets the GPS to start the communication.

```
softserial.begin(9600);
```

Once this is done, we can go to the void loop() method and add two float variables "float latd, lond;" (*Note* It is very important to make these two variables a float value because we want to be as accurate as possible about our location). Next we create a while loop that will continuously retrieve the data when it's available and store it in the variables we created.

```
float latd, lond;
while (softserial.available())
 {
   gpsplus.encode(softserial.read()); //This line of code will first read the data from the serial ports from softserial then after the encode function will take the GPS readings from the serial ports and encodes it.

   latd = gpsplus.location.lat(); // stores the latitude values
   lond = gpsplus.location.lng(); // stores the longitude values
 }
```

Finally, right below the while loop we can print out the values for the GPS to the users to see. Note the ", 6" tells the Arduino to print up to 6 decimal places. This is to display an accurate GPS reading.

```
Serial.print(latd, 6);
Serial.print(" ");
Serial.print(lond, 6);
```

Up to here your Arduino code should now look like this:

```
#include <MPL3115A2.h>
```

```
#include <HTU21D.h>
#include <Wire.h>
#include <TinyGPS++.h>
#include <SoftwareSerial.h>
HTU21D     humidity;
MPL3115A2  pressure;
TinyGPSPlus  gpsplus;
SoftwareSerial softserial(5, 4); // pin 5 for rx and 4 for tx

// variables for wind speed and direction calculation.
volatile int contacts = 0; long lastCalcTime = 0;

// interrupt service routine for when the magnets contact each other on the anemometer
// the interrupt occurs when the switches contact. each time this happens the counter
goes up
void isrSpeed()
{
  // checking if time passed between now and the last calculation is greater than 15 ms
  // to prevent the program from reading the bouncing on the switches
  if (millis() - lastCalcTime > 15)
  contacts++; // increment counter
}

void setup() {
  // put your setup code here, to run once:
  Wire.begin();
  Serial.begin(9600); // for prinitng
  softserial.begin(9600); // for listening to GPS
  humidity.begin();
  pressure.begin();
  pressure.enableEventFlags();
  pressure.setModeBarometer();
  pressure.setModeActive();
  pinMode(A1, INPUT); // this is the pin for the light sensor
  pinMode(A3, INPUT); // pin for operating voltage used for light calculation
```

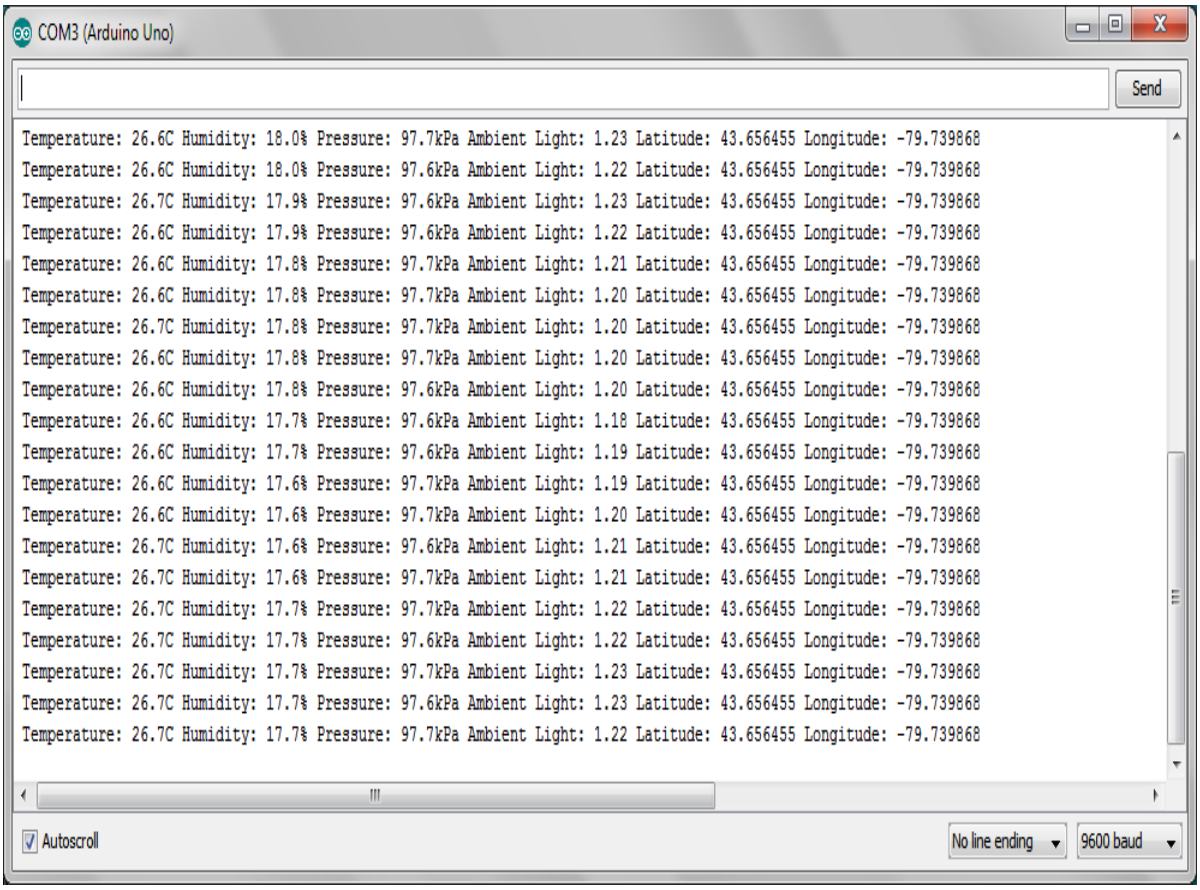
```
  pinMode(3, INPUT_PULLUP); // pin for the windspeed input, pull up resistor needed,
hence input_pullup

  // adding the interrupts to the program and enabling them
  attachInterrupt(1, isrSpeed, FALLING); // priority 1, function for interrupt, act when
value goes to low
  interrupts();
}

void loop()
{
  // read values
  float humd = humidity.readHumidity();
  float temp = pressure.readTemp();
  float pres = pressure.readPressure();
  float lght = readLight();
  float latd, lond;
  float wspd = calculateWindSpeed();
  int wdir = calculateWindDirection();

  // get data from gps when available and store coordinates
  while (softserial.available())
  {
    gpsplus.encode(softserial.read());
    latd = gpsplus.location.lat();
    lond = gpsplus.location.lng();
  }

  // printing values
  Serial.print(temp, 1);
  Serial.print(" ");
  Serial.print(humd, 1);
  Serial.print(" ");
  Serial.print(pres / 1000 , 1); // divide by 1000 to get in kilo Pascals
  Serial.print(" ");
  Serial.print(lght, 2);
```

```
  Serial.print(" ");
  Serial.print(latd, 6);
  Serial.print(" ");
  Serial.print(lond, 6);
  Serial.print(" ");
  Serial.print(wspd);
  Serial.print(" ");
  Serial.print(wdir);
  Serial.print(" ");

  Serial.println(); // print a new line for the next values
  delay(1000); // wait 1 second before printing new values
}

float readLight()
{
  float operVolt = analogRead(A3); // get the operating voltage of the weather shield
  float lightSen = analogRead(A1); // get input ratiometric voltage for the light sensor
  operVolt = 3.3 / operVolt;       // divide the full scale voltage by the operating voltage
  lightSen *= operVolt;            // multiply this voltage with the light sensor ratiometric
voltage to get ambient light result
  return(lightSen);
}

float calculateWindSpeed()
{
  float timeChange = (millis() - lastCalcTime) / 1000
  float windSpeed = (((float)contacts / timeChange) * 1.492) * 1.60934
  contacts = 0;
  lastCalcTime = millis();
  return windSpeed;
}

int calculateWindDirection()
{
  unsigned int analogValue;
```

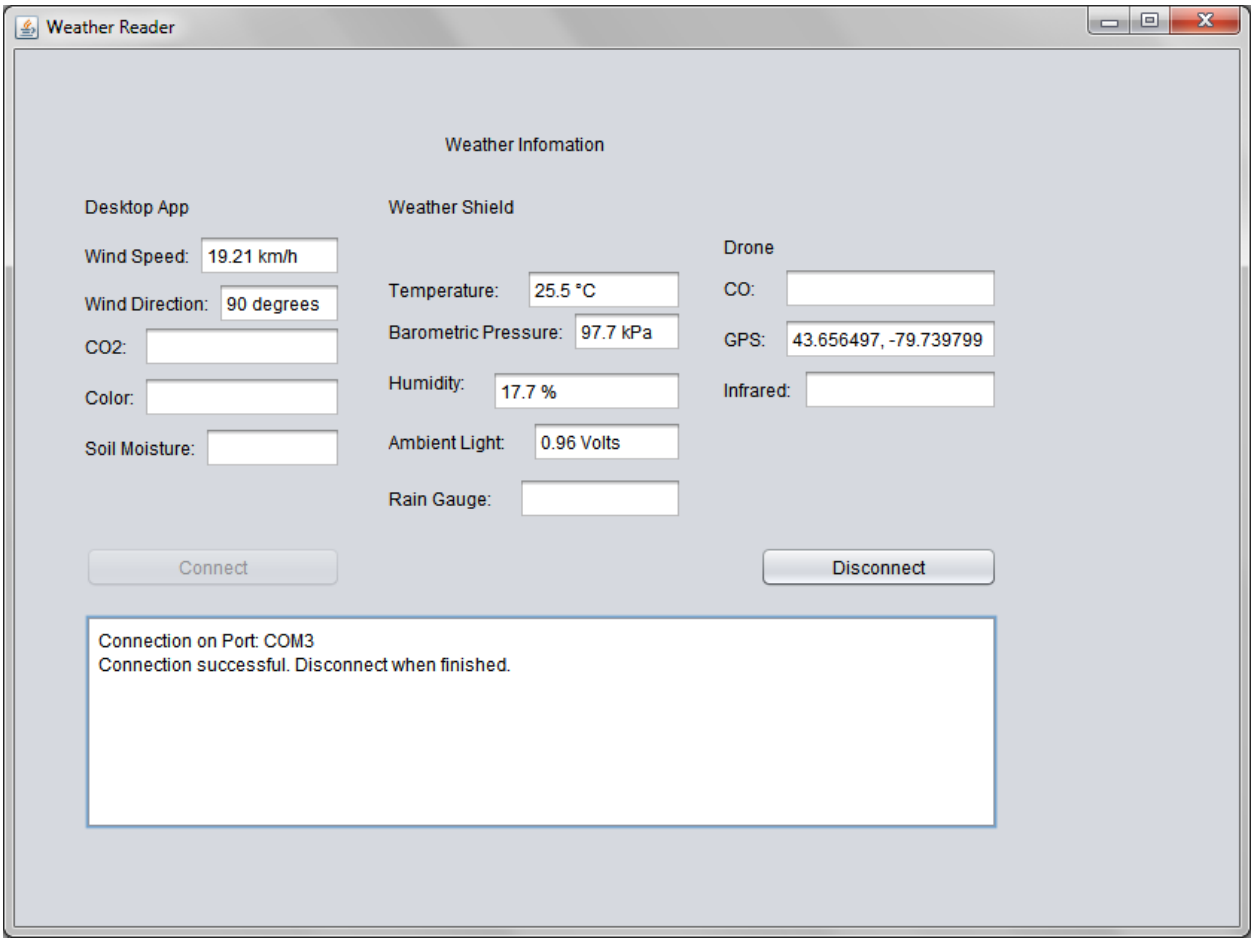
```
  analogValue = analogRead(A0);

  if (analogValue < 414) return (90);  // east
  if (analogValue < 508) return (135); // south-east
  if (analogValue < 615) return (180); // south
  if (analogValue < 746) return (45);  // north-east
  if (analogValue < 833) return (225); // south-west
  if (analogValue < 913) return (0);   // north
  if (analogValue < 967) return (315); // north-west
  if (analogValue < 990) return (270); // west

}
```

Notice: The GPS is not a very strong GPS. For best results hold the GPS near a window or use it outside then upload the program. You may see 0.000000 0.000000 at first but after a while it will show up a value. If after a while it does not change, try holding it up to a window or place it outside then upload the code. Make sure the serial switch on the weather shield is set to soft.

Your Arduino output may like this without the labels: Latitude and longitude are printed to the far right.



The Java code you copied and pasted already has the code to get the GPS working. Now connect the board, open the Java program and then try it out.


Weather Reader
Weather Infomation
Desktop App
Wind Speed: 19.21 km/h
Wind Direction: 90 degrees
CO2:
Color:
Soil Moisture:
Weather Shield
Temperature: 25.5 °C
Barometric Pressure: 97.7 kPa
Humidity: 17.7 %
Ambient Light: 0.96 Volts
Rain Gauge:
Drone
CO:
GPS: 43.656497, -79.739799
Infrared:
Connect
Disconnect
Connection on Port: COM3
Connection successful. Disconnect when finished.

WEEK 8

Objectives for this week:

- Design a simple anemometer with the *.stl files provided
- 3D printing an anemometer

WEEK 9

Objective for this week:

- Design Showcase

MAKERSPACE CREATIVE HUB

Drop In Schedule

@ Four Corners Branch Library
65 Queen Street East, Brampton

FEB 2

Explore Architectural Design
Monday, February 2nd, 2:30 - 5:30pm
The community is invited to come and learn about Architectural Design.

FEB 3

Come Play with Makey Makey (Arduino)!
Tuesday, February 3rd, 4:00 - 7:00pm
NEW!
All ages are welcome to come and play with Arduino! Make music with bananas or video game controllers with play doh!

FEB 4

Discover 3D Printing and Modeling
Wednesday, February 4th, 4:00 to 7:00pm
The community is invited to explore 3D printing and Modeling. Beginner workshop starts at 4pm.

FEB 5

Residential Renovation Workshop
Thursday, February 5th, 5:00 - 8:00pm
Calling all Renovators! The community is invited to share - and solve - your renovation challenges.

WEATHER STATION DESIGN NEW!

Workshop Series: Saturdays, Feb 7 to Mar 28
11am to 2pm, Showcase on March 31

This new workshop series is an exciting opportunity for high school students or interested adults to design a Weather Station Receiver system. You will be provided with a Weather Station Receiver Parts Kit (Arduino powered), then participate in a 9-week Workshop series to 'design-build-test' your own weather station. The workshop series will include the design and 3D printing of a small scale weather station.

Ages 14+ Free of charge. Register at Library or weatherstation.eventbrite.com

LEGO Robotics Mindstorm NEW!

Workshop Series: Sundays Feb 8 to Mar 1st
1:30 to 4:30 pm, Showcase on March 15

LEGO Mindstorms is a terrific, easily accessible, introduction to robotics, and frankly just a whole lot of fun! A Lego kit will be available at the workshop for you to work with and every week you and your small team will learn how to build and program your own Robot. At the showcase, invite your family and friends to see the Robot competition and help celebrate Engineering Month.

Ages 12 – 18 yrs. Free of charge. Register at Library or LegoRoboticsFeb2015.eventbrite.com

Week 11

Working with the Arduino Due

Up until now we have been working with the Arduino Uno boards. Now let's move on to the Arduino Due boards.

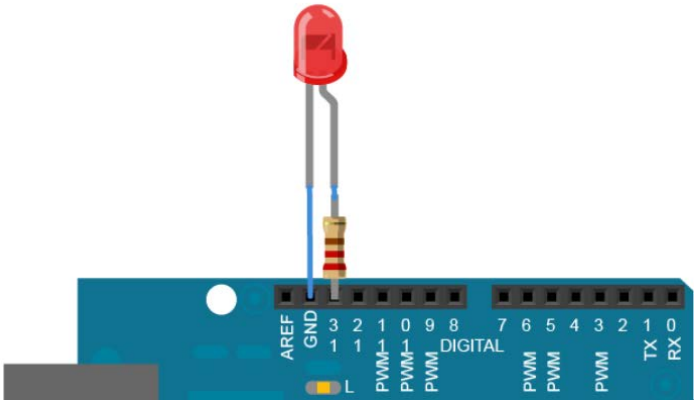
Due vs. Uno

The Arduino Uno is a popular board. There are lots of publicly available libraries and code that can be used. However the due does not have a lot of publicly available code for it. Users of the Due might have to create their own libraries such as us for example. The goal from now on is to get the weather station working with the Due. Unfortunately not only are there not much open source code and libraries out there but there are not much components for it to if I'm not mistaken so consequentially users might have to make their own boards. The Sparkfun Weather Shield we used earlier was made only for the Arduino Uno so if we want to measure humidity, pressure, temperature, or wind speed, we would have to make a brand new board for it, which eventually we will. Let's focus on the basic differences between the Uno and the Due.

The Due's processor is an ARM processor. ARM processors can be found in applications such as smart phones for example. Specifically its processor is the AT91SAM3X8E. The operating voltage on the Due is 3.3 V where the Uno it's 5 V. The processor on the Due is faster too; 84 MHz compared to the Uno's 16 MHz. The Due has more pins than the Uno. It has 12 analog pins and 54 digital pins where the Uno has 6 analog and 14 digital. The Due does not have and EEPROM, but it does have 96 KB of SRAM and 512 KB of Flash memory. The Uno has 1 KB of EEPROM memory 2 KB SRAM and 32 KB of Flash memory. You can clearly see if you were to write a program for the Uno it can't be too big. The large memory in the Due allows for bigger/more complex programs. Also if you have a Due notice that it has two USB ports native and programming. Let's use the programming one as we are going to be writing programs to the Due. Let's get started.

Your first Arduino Due program

In this section of the manual we will be starting the upgraded version of the Ardunio. This Microprocessor is called the Due a far stronger beast than its predecessor the Uno. We will write our first program on blink LED lights with the due. First things first is to get a LED light and plug the short end of the LED to ground and the long end can be on any pin number. In this example we will be using pin 13.



After you have completed this task we will go onto the programming part of it. You first need to set up the board so first you need to go to Tools -> Board -> Boards Manager and select the Arduino SAM boards (32-bits ARM Cortex-M3) and make sure to install the correct version according to your Arduino IDE; after completed go to Tools -> Board -> Arduino Due (Programming Port). Lastly make sure the correct communication ports are properly selected; Tools-> Ports. You are now ready to write the code to complete this exercise!

First things first we will write our setup function:

```
1  void setup() // this function will initialize the values we will need for the program
2  {
3    pinMode(13,OUTPUT); // this line initializes that pin 13 will be the output
4  }
```

Lastly we write out main program in a loop function. This loop will run forever until the user hits the reset button on the Due board:

```
6  void loop() // this function will continuously loop running the main program
7  {
8    digitalWrite(13, HIGH); // this line of code will turn on the LED
9    delay(1000); // will wait for 1 second (measured in miliseconds)
10   digitalWrite(13, LOW); // this will turn the LED light off
11   delay(1000); // wait for 1 second
12 }
```

A general explanation of what's happening here is that we are supplying 5 volts to pin 13 when we say "HIGH" which will light up the LED then we delay it for 1 second and then supply 0 volts (which is "LOW") thus will result in turning off the LED then we delay it once more for 1 second and then the loop will jump back to the top of the function and start the process all over again. (Note: instead of writing out the pin number "13" try putting it in a variable by declaring it before the setup function).

```
const byte ledpin = 13;

void setup(){
        pinMode(ledpin, OUTPUT);
        // etc
}
```

Exercise: At this point you may have realized that this code is no different than the code you did for the Arduino Uno in week 1. However we are just getting started and soon we'll find differences. Since you should be familiar already with this code make the LED stay off for half of a second. Then try keeping on for half a second and off for 1/4 of a second. Note the value you put is in milliseconds. 1000 ms = 1 second.

Week 12

Infrared Light Sensor

This week we will be discussing about installing the infrared light sensor on the Arduino Due. The infrared light sensor is converts infrared to voltage. So that means the measurements we will be getting will be in voltage. This will be a two part setup; first we do the physical set up with the material list provided, then we do the coding.

Part 1: Physical Setup

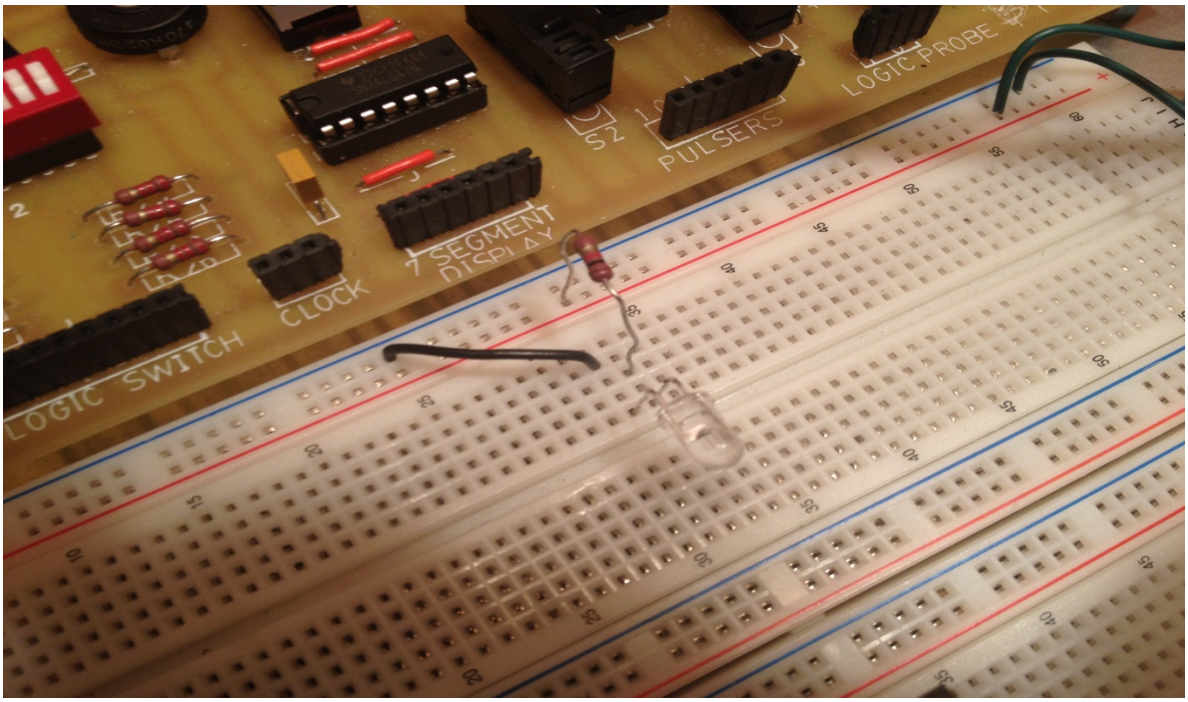
Material List
- 1 Perf Board
- 1 TSOP 38338 Infrared Light Sensor (datasheet)
- 1 LTE-5208A Infrared Emitter (datasheet)
- 3 different colored jumper wires (3 different colors for clarity)
- 1 Arduino Due Board
- Wire cutters
- Lead cutters
- USB cable (hook the Arduino Due to the PC)
- 1 capacitor (1uf)
- 2 resistors (330 ohms and 1 kohm)
- 1 power supply
- Mulitmeter (optional)
- 2 Solderless breadboards

Now we got the material list out of the way let's do the physical setup for it.

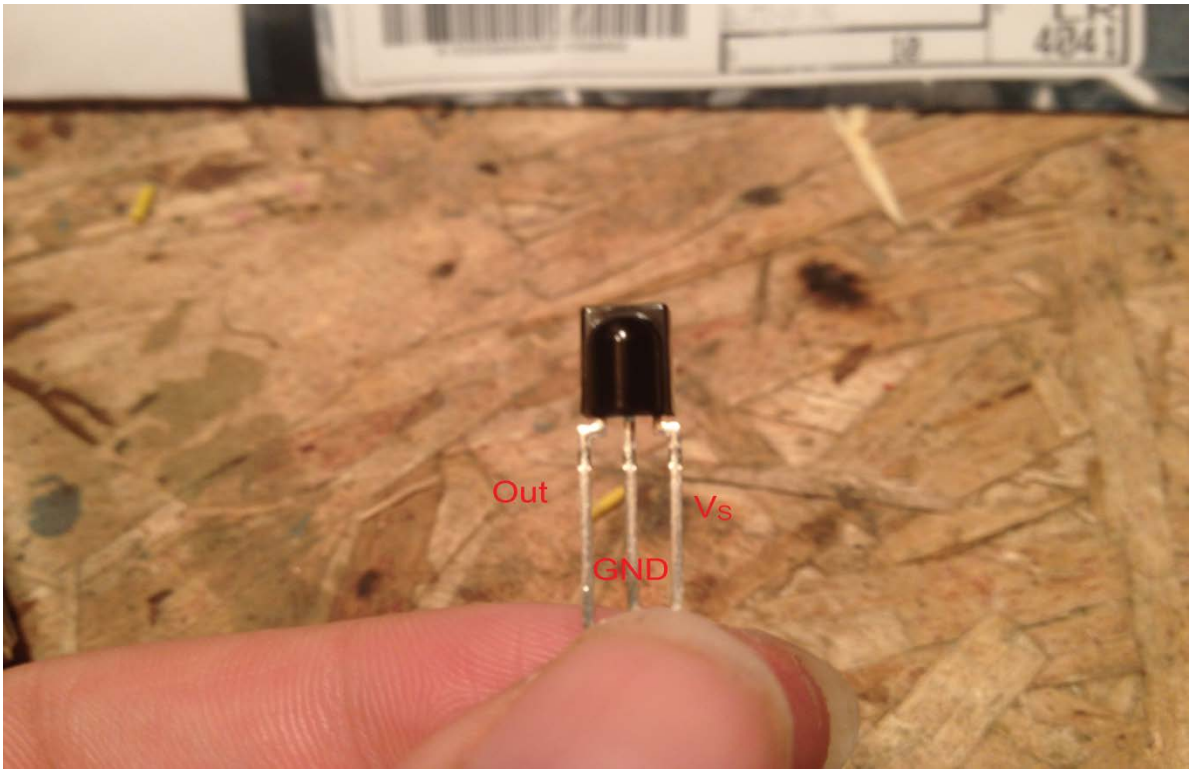
You need to have a power supply for the IR light. If you can, use the power supply for both the sensor and the emitter. In this tutorial a fabrication kit was used. It's pretty much a portable power supply.

On one solderless breadboard place the IR emitter near the middle of the board and bend the top of the IR emitter so it faces toward you. Connect a 1 kohm resistor from the long

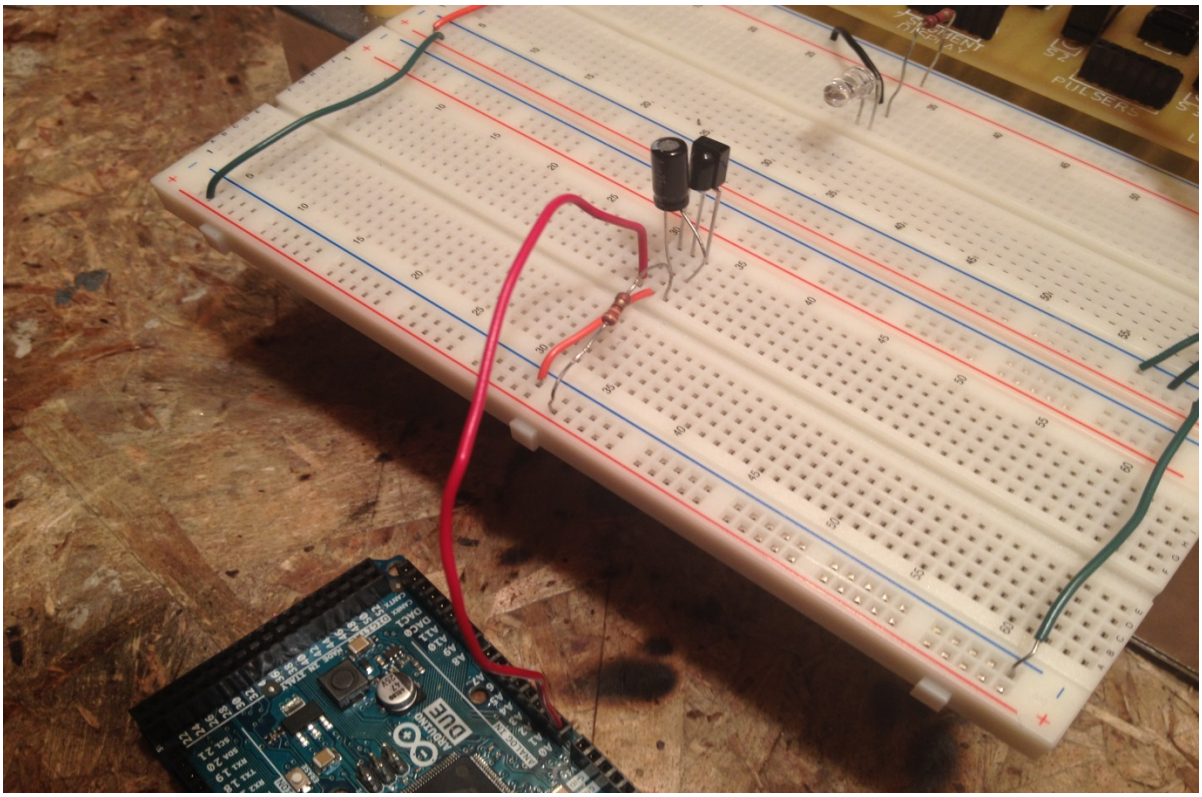
lead to 5V on the power supply. Set the power supply to 5V if needed and connect a lead to the resistor from the power supply. Connect another lead from – on the power supply to the short pin on the IR sensor.



Now for the IR sensor: First let's take a look at the pin configurations. Looking at the front of the sensor with the round part, the output to the Arduino is the pin on the left, the pin in the middle goes to ground, and the pin on the right connects to the 5V from the power supply.



Connect the sensor to the second breadboard with the "bump" part of the sensor facing you so that way the flat part is facing the emitter. Next connect a polarized 1uf capacitor from the V_S pin to the ground pin. The negative pin on the capacitor is the short pin. This pin needs to be connected to the ground pin. Next connect a 330 ohm resistor from the V_S to a lead that connects to 5V on the power supply. Connect a lead from the power supply negative to the ground pin (GND). Now connect a wire from the output pin to pin A0 on the Arduino Due.



Now let's write the code for the Arduino.

Part 2: Coding the infrared light sensor

First things first are we start off by writing our setup function.

```
1  void setup()
2  {
3    Serial.begin(9600);
4  }
```

This here will setup printing the data and printing rate. Next we will write out our loop. Within our loop we want to get the IR volt from our function and print it out to the user.

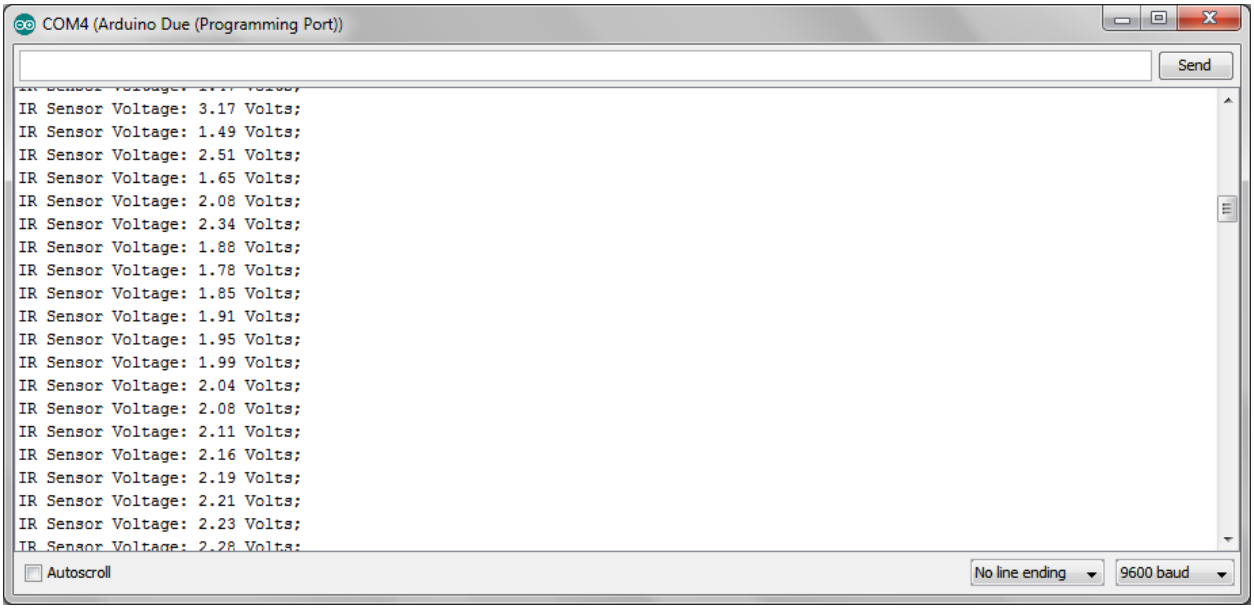
```
void loop() {
```

```
  float IRValue = getIRVolts(); // Creates a floating point precision variable and stores the results in it
  Serial.print("IR Sensor Voltage: "); // prints to the user
  Serial.print(IRValue, 1);
  Serial.print(" Volts; ");
  Serial.println();
  delay(1000); // delay for 1 second (1000 milisecond = 1 second)
}
```

Finally, we will create the function that will get the IR voltages. The IR voltage is determines by how much heat there is around the device. The more heat there is exposed to the sensor, the voltage levels starts to decrease.

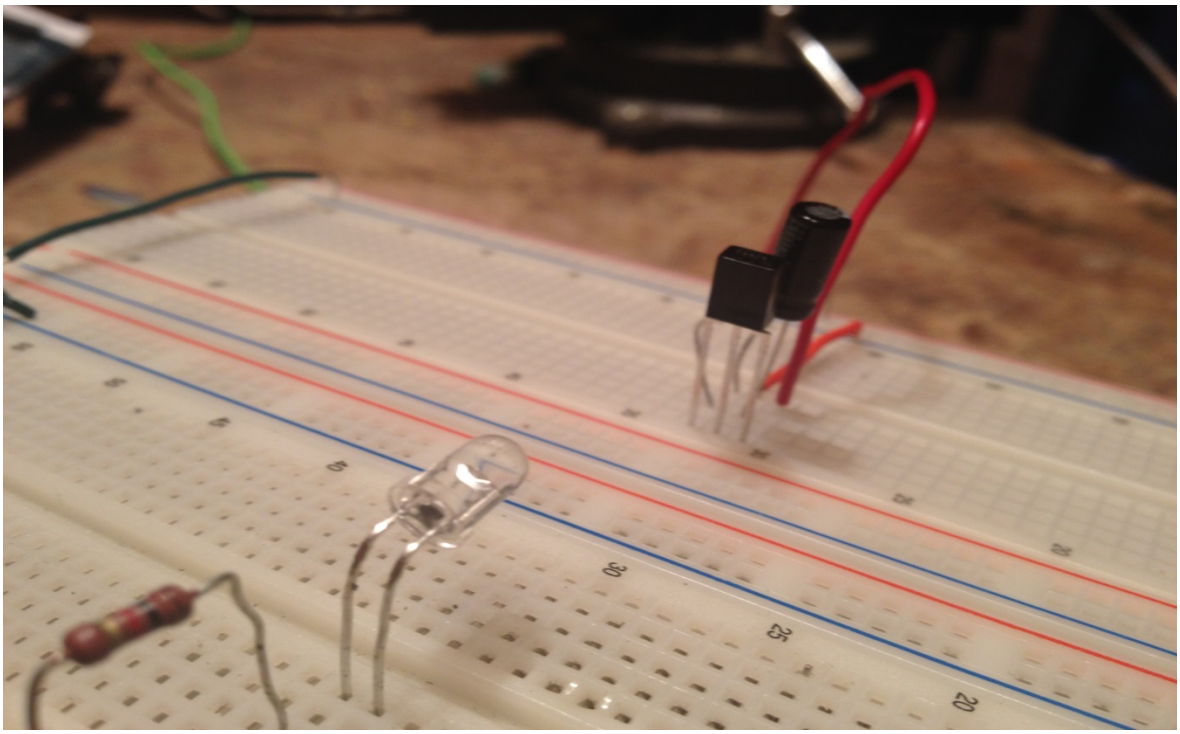
```
// we are calculating how much voltage is going into the analog port
float getIRVolts()
{
  int inputValueIR = analogRead(A0); // read raw data on port A0

  // voltage on port = (max voltage on due / max raw data value on port) * current raw data value on port
  float IRVolts = (3.3 / 1023) * inputValueIR; // caluclating voltage
  return IRVolts;
}
```

Upload the program to the Due. Here is a sample output.



When the infrared sensor picks up infrared, the voltage that is output should increase. Try turning off any lights in the room so the IR sensor can detect the infrared better. Now try putting you hand in front of the sensor and see what happens. You may need to use a multimeter in order to see the voltage change more accurately. Connect the + lead of

the multimeter to the output pin of the sensor and connect the – to the ground pin. Set the multimeter to measure voltage and now try turning off the lights and using your hand to block the sensor.



Week 13

Carbon Monoxide Sensor

The CO sensor we will be using this week is the MQ-9B. The MQ sensors are a series of sensors that measure gasses and output a voltage based on the concentration of gas. The voltage reading is the voltage across the load resistor which is connected from the output to the ground. This voltage along with the voltage input into the circuit for the sensor and the resistance of the load resistor is used in a formula to find the resistance of the sensor which is then used to find out what the concentration of the gas is in parts-per-million.

Along with the MQ-9B sensor we will also use the Pololu mini board for the CO sensor. Before we go and connect the sensor there are some important things to learn first.

Connection

It may not be important for this device but some sensors need to be connected in a specific way. They may have A pins and B pins and need to be connected to the A and B ports respectively on the mini IC board. If so it is important these pins are connected properly or the device can be instantly damaged or destroyed. Some devices use a lot of current so it is not recommended to connect the device directly to the Arduino if it may use >150 mA of current.

Heater

Some devices such as this device need to alternate between voltages for the heater for certain periods of time for best results. This device needs the heater voltage alternate between 5 V and 1.5 V for 60 seconds and 90 seconds respectively for the heater to work properly and to measure different gasses, according to the store page and interpretation of the data sheet. CO is measured during the time the heater is using 1.5 V.

Preheat/Burn-in

The gas sensors have a burn-in period for which the heater must be properly powered for a certain time so the readings can settle. It is important to have the device powered up to the preheat time specified on the data sheet so that way the results are reliable. These device take serval hours to preheat, some even take a couple of days, such as the case with the sensor we will be using.

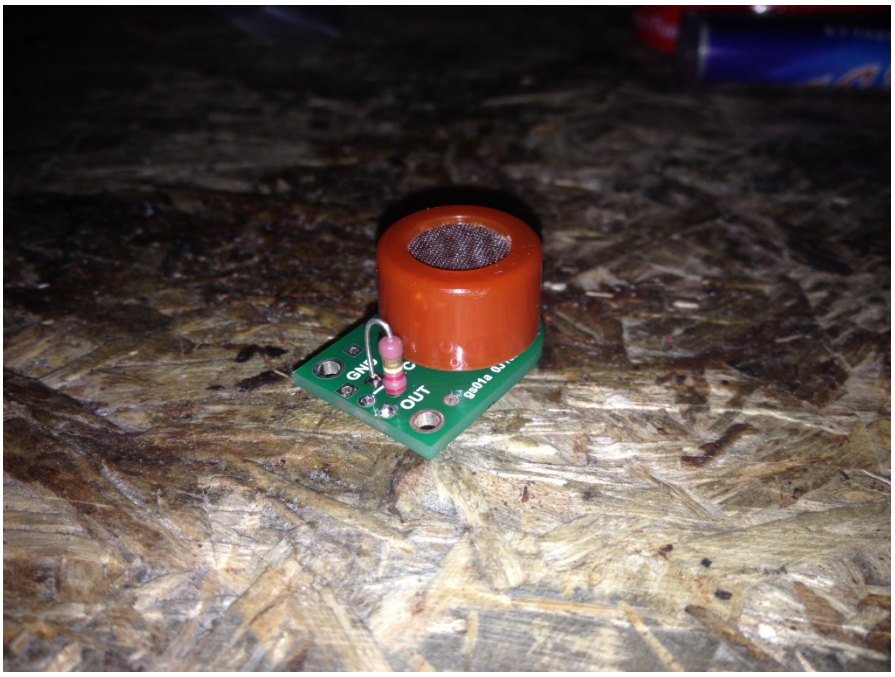
Load Resistor

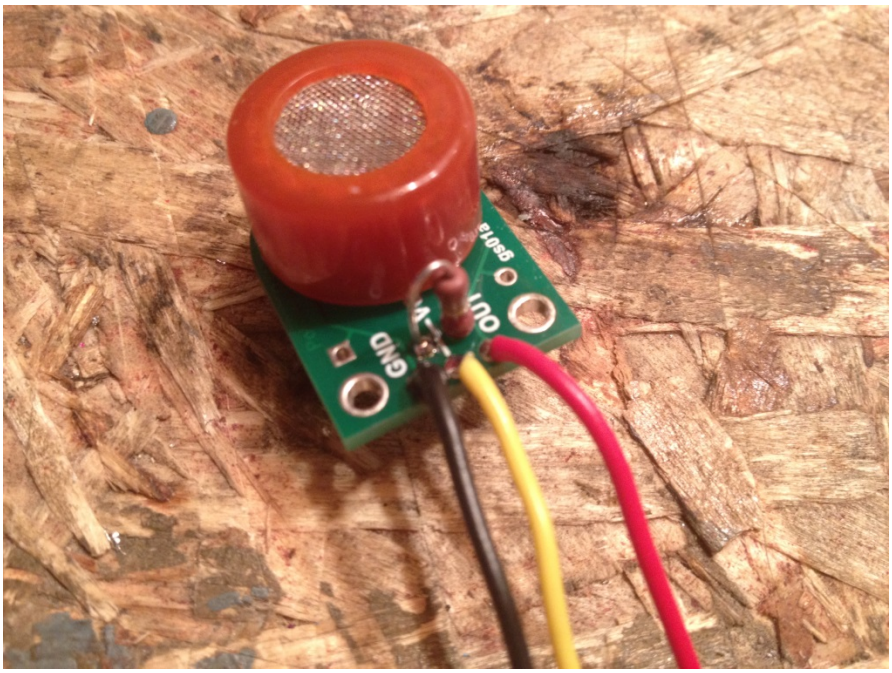
You will need a load resistor to calibrate the device sensitivity and for the output results. It can be any resistor between 2 kOhm and 47 kOhm but the lower the resistance the less sensitive the device is and the higher the resistance the less accurate the device is when it is measuring high concentrations. We will be using 22kOhm for the resistance.

Datasheet

Last but not least it is also important to refer to the datasheets. The datasheet for the MQ-9 sensor can be found here.

Now let's get started. First place the sensor into the mini IC board and solder the pins. Since the pins do not appear to be marked A or B, it may be placed into the board in any way. Next solder in a resistor from the small hole near ground to the small hole near the output. See the picture below for reference. Next cut three long wires of different colour to the ground hole, VCC hole, and output hole.





(more will be added later)

Week 14

Rain Gauge

In this chapter we will be going back to the Arduino Uno and the Sparkfun weather shield. Make sure you have those before attempting this section. You will also need a rain gauge. If you have the anemometer from the section where wind speed and direction was covered, you will only need the rain gauge part. Once you have these items let's move on.

This getting the rain gauge to work is very similar to calculating wind speed as you will need to use interrupts. However, it's much simpler because other than printing the values and setting up the pin and adding the interrupt, the only challenge you will need to do is just have an interrupt service routine to increment a counter every time the switch closes, when water causes the balance in the sensor to tip. Recall for the wind speed we also had to take that counter and use it for a calculation. Here all we care about is the value of the counter. Now of course we need to account for switch bounce so we don't get unusual readings.

Open up the Arduino .ino file that has the measurements such as temperature, humidity, wind speed, wind direction etc. that you have used to get readings in the Java program. If you don't recall how to do interrupt service routines, refer to the section in this manual the covers how to calculate wind speed as interrupts were covered in that section.

Coding:
What you need to add to the program are two variables near the top section (where all other global variables are declared) that stores the rain count in mm or inches and another variable that stores the last time the value was calculated. Note that both variables are going to be modified in the interrupt service routine. Also note that the count variable will store a decimal number.

Then you need to write the function that handles the interrupt. Inside the function you will check if the last time the value was checked is greater than 15 ms and if so you will simply increment the counter by 0.011 for inches or 0.02794 for millimeters and then

store the current time into the variable that stores the last time checked by using the millis() function.

A word about if statements: you start with if and inside the brackets you put the condition. In this case you check if the current time subtracted by the last checked time is greater than 15. Use > symbol for greater than. Note that you can also check for equality in these statements as well as checking if values are greater or less than but we will just be checking if the value is greater than 15. After this section your code should be enclosed in square brackets.

Example syntax:

```
if(condition)
  {
    //code
  }
```

Note that if there is only one line of code that is executed when the condition is true, the brackets are not necessary. However it's good practice to always have brackets.

Next add pin 2 and an input with a pullup resistor added just like you did with the pin that reads the anemometer. Note that pin 2 is the pin the rain gauge is connected to.

Finally the last thing is too add the interrupt function with priority of 0 and triggers when the value falls. Then print the variable that calculates rain fall.

Answer:
Copy these separately and paste into the proper sections or you can copy the entire code below this part.

Paste to section of code where global variables are declared.

```
// variables for rain guage
volatile float rainLevel = 0; // variable where the rain count in mm is stored
volatile long lastRainMeasTime = 0; // variable that stores when the last time the rain was measured for switch debounce purposes
```

Paste after the function that calculates wind speed.

```
// interrupt service routine for when the switch closes on the rain guage due to water
// when switch is closed increment the rain counter
void isrRain()
{
  // check when the last time the rain was measured is greater than 15 ms to
  // ignore switch bounce, if true increment the rain counter and store current time
  if (millis() - lastRainMeasTime > 15)
  {
    rainLevel += 0.2794; // increment the counter; 0.2796 mm is enough rain to make the
contact close
    lastRainMeasTime = millis(); // store time for next time the switch closes
  }
}
```

Paste into void setup() near where the interrupt for the wind speed was added.

```
pinMode(2, INPUT_PULLUP); // pin for the rain gauge

  // adding the interrupts to the program and enabling them
  attachInterrupt(0, isrRain, FALLING); // add interrupt for rain gauge, priority 0, for
when the value goes to low
```

Paste into void loop() where variables were printed

```
Serial.print(rainLevel, 2);
Serial.print(" ");
```

The entire code up to this point

That's it. This is what the code should now look like:

```
#include <MPL3115A2.h>
#include <HTU21D.h>
#include <Wire.h>
```

```
#include <TinyGPS++.h>
#include <SoftwareSerial.h>
HTU21D     humidity;
MPL3115A2  pressure;
TinyGPSPlus  gpsplus;
SoftwareSerial softserial(5, 4); // pin 5 for rx and 4 for tx

// volatile variables are modified during interrupts

// variables for wind speed and direction calculation.
volatile int contacts = 0; // track how much times the switches/magnets on the
anemometer close
long lastCalcTime = 0; // variable that stores when the last time the wind speed was
calculated for calculation purposes

// variables for rain guage
volatile float rainLevel = 0; // variable where the rain count in mm is stored
volatile long lastRainMeasTime = 0; // variable that stores when the last time the rain was
measured for switch debounce purposes

// interrupt service routine for when the magnets contact each other on the anemometer
// the interrupt occurs when the switches contact. each time this happens the counter
goes up
void isrSpeed()
{
  // checking if time passed between now and the last calculation is greater than 15 ms
  // to prevent the program from reading the bouncing on the switches
  if (millis() - lastCalcTime > 15)
  contacts++; // increment counter
}

// interrupt service routine for when the switch closes on the rain guage due to water
// when switch is closed increment the rain counter
void isrRain()
{
  // check when the last time the rain was measured is greater than 15 ms to
```

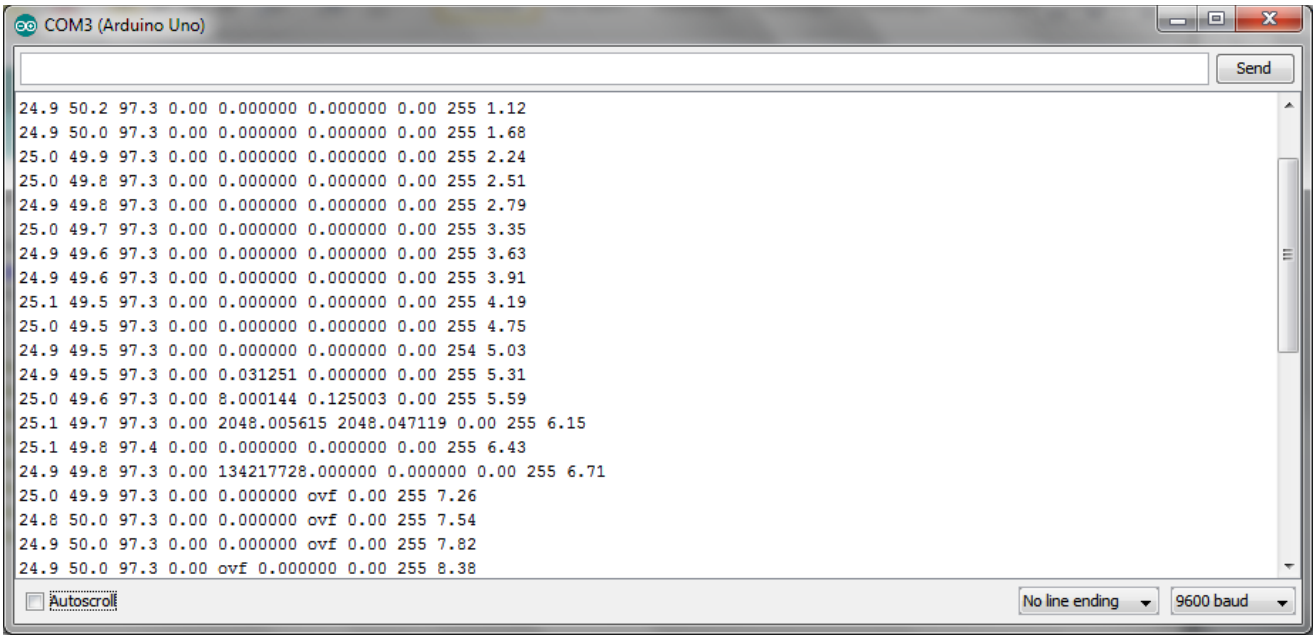
```
  // ignore switch bounce, if true increment the rain counter and store current time
  if (millis() - lastRainMeasTime > 15)
  {
    rainLevel += 0.2794; // increment the counter; 0.2796 mm is enough rain to make the
contact close
    lastRainMeasTime = millis(); // store time for next time the switch closes
  }
}

void setup() {
  // put your setup code here, to run once:
  Wire.begin();
  Serial.begin(9600); // for prinitng
  softserial.begin(9600); // for listening to GPS
  humidity.begin();
  pressure.begin();
  pressure.enableEventFlags();
  pressure.setModeBarometer();
  pressure.setModeActive();
  pinMode(A1, INPUT); // this is the pin for the light sensor
  pinMode(A3, INPUT); // pin for operating voltage used for light calculation
  pinMode(3, INPUT_PULLUP); // pin for the windspeed input, pull up resistor needed,
hence input_pullup
  pinMode(2, INPUT_PULLUP); // pin for the rain guage

  // adding the interrupts to the program and enabling them
  attachInterrupt(0, isrRain, FALLING); // add interrupt for rain guage, priority 0, for
when the value gos to low
  attachInterrupt(1, isrSpeed, FALLING); // priority 1, function for interrupt, act when
value goes to low
  interrupts();
}

void loop()
{
  // read values
```

```
float humd = humidity.readHumidity();
float temp = pressure.readTemp();
float pres = pressure.readPressure();
float lght = readLight();
float latd, lond;
float wspd = calculateWindSpeed();
int wdir = calculateWindDirection();

// get data from gps when available and store coordinates
while (softserial.available())
{
  gpsplus.encode(softserial.read());
  latd = gpsplus.location.lat();
  lond = gpsplus.location.lng();
}

// printing values
Serial.print(temp, 1);
Serial.print(" ");
Serial.print(humd, 1);
Serial.print(" ");
Serial.print(pres / 1000 , 1); // divide by 1000 to get in kilo Pascals
Serial.print(" ");
Serial.print(lght, 2);
Serial.print(" ");
Serial.print(latd, 6);
Serial.print(" ");
Serial.print(lond, 6);
Serial.print(" ");
Serial.print(wspd);
Serial.print(" ");
Serial.print(wdir);
Serial.print(" ");
Serial.print(rainLevel, 2);
Serial.print(" ");
```

```
  Serial.println(); // print a new line for the next values
  delay(1000); // wait 1 second before printing new values
}

float readLight()
{
  float operVolt = analogRead(A3); // get the operating voltage of the weather shield
  float lightSen = analogRead(A1); // get input ratiometric voltage for the light sensor
  operVolt = 3.3 / operVolt;       // divide the full scale voltage by the operating voltage
  lightSen *= operVolt;            // multiply this voltage with the light sensor ratiometric
voltage to get ambient light result
  return(lightSen);
}

float calculateWindSpeed()
{
  // get the time change by taking the current time (ms) and subtracting it by the last
time the speed was calculated
  float timeChange = (millis() - lastCalcTime) / 1000;              // and then converting it
into seconds

  // caclulating wind speed by taking the number of contacts the magnets/switches made
divided by the time since we last made the speed calculation
  float windSpeed = (((float)contacts / timeChange) * 1.492) * 1.60934;  // and then
multipling by the speed per click per second and coverting to km/h
  contacts = 0;                                                  // reset the switch contact count and
count again for new calculation
  lastCalcTime = millis();  // store the new time in the last calculated speed time variable
  return windSpeed;
}

int calculateWindDirection()
{
  unsigned int analogValue;
  analogValue = analogRead(A0); // read value from the analog port A0, where the reading
from the wind speed comes from
```

```
  /* when the input analog value gives a certain reading, return the specific direction in degrees, see weather meters datasheet
   *
https://www.sparkfun.com/datasheets/Sensors/Weather/Weather%20Sensor%20Assembly..pdf
   * source: uno_weathershield by Hang and Jason
   * we are assuming that we only care about 8 directions since we can't precisely get all 360 readings from the anemometer */

  if (analogValue < 414) return (90);  // east
  if (analogValue < 508) return (135); // south-east
  if (analogValue < 615) return (180); // south
  if (analogValue < 746) return (45);  // north-east
  if (analogValue < 833) return (225); // south-west
  if (analogValue < 913) return (0);   // north
  if (analogValue < 967) return (315); // north-west
  if (analogValue < 990) return (270); // west
}
```

Connect the board and rain gauge and upload the program. Put water into the rain gauge or tilt the gauge back and forth and observe the results. The measurement of the rain is the value on the far right. Note that if you had copied the above code this value is in millimetres.


COM3 (Arduino Uno)
Send
24.9 50.2 97.3 0.00 0.000000 0.000000 0.00 255 1.12
24.9 50.0 97.3 0.00 0.000000 0.000000 0.00 255 1.68
25.0 49.9 97.3 0.00 0.000000 0.000000 0.00 255 2.24
25.0 49.8 97.3 0.00 0.000000 0.000000 0.00 255 2.51
24.9 49.8 97.3 0.00 0.000000 0.000000 0.00 255 2.79
25.0 49.7 97.3 0.00 0.000000 0.000000 0.00 255 3.35
24.9 49.6 97.3 0.00 0.000000 0.000000 0.00 255 3.63
24.9 49.6 97.3 0.00 0.000000 0.000000 0.00 255 3.91
25.1 49.5 97.3 0.00 0.000000 0.000000 0.00 255 4.19
25.0 49.5 97.3 0.00 0.000000 0.000000 0.00 255 4.75
24.9 49.5 97.3 0.00 0.000000 0.000000 0.00 254 5.03
24.9 49.5 97.3 0.00 0.031251 0.000000 0.00 255 5.31
25.0 49.6 97.3 0.00 8.000144 0.125003 0.00 255 5.59
25.1 49.7 97.3 0.00 2048.005615 2048.047119 0.00 255 6.15
25.1 49.8 97.4 0.00 0.000000 0.000000 0.00 255 6.43
24.9 49.8 97.3 0.00 134217728.000000 0.000000 0.00 255 6.71
25.0 49.9 97.3 0.00 0.000000 ovf 0.00 255 7.26
24.8 50.0 97.3 0.00 0.000000 ovf 0.00 255 7.54
24.9 50.0 97.3 0.00 0.000000 ovf 0.00 255 7.82
24.9 50.0 97.3 0.00 ovf 0.000000 0.00 255 8.38
Autoscroll
No line ending
9600 baud